Titre: Title:	Virtual Machine Flow Analysis Using Host Kernel Tracing
Auteur: Author:	Hani Nemati
Date:	2019
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Nemati, H. (2019). Virtual Machine Flow Analysis Using Host Kernel Tracing [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/3902/

UNIVERSITÉ DE MONTRÉAL

VIRTUAL MACHINE FLOW ANALYSIS USING HOST KERNEL TRACING

HANI NEMATI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
MAI 2019

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

VIRTUAL MACHINE FLOW ANALYSIS USING HOST KERNEL TRACING

présentée par: NEMATI Hani
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. KHOMH Foutse, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. ALOISE Daniel, Ph. D., membre
M. KHENDEK Ferhat, Ph. D., membre externe

DEDICATION

*To My Loving **Parents**, specially my Mom, Without the inspiration, drive, and support that you have given me, I might not be the person I am today.*

*To My Lovely **Mahsa**, You changed my world since you entered into my life. The change is not simple and subtle. You added colors, meaning, and happiness in my life. I love you.*

*To **Ayden**, You are a special gift from the heavens. Your smile warms my heart. I love you, today and forever,...*

ACKNOWLEDGEMENTS

I would like to thank,

Michel Dagenais, for endless support and enthusiasm, for always being so positive, always believing in me and my capacity. and for creating such a lovely milieu in the lab. I can not imagine a better supervisor than you.

Seyed Vahid Azhari, I admire your philosophy on hard work but at the same time very permissive attitude. I have had much fun finding out new things with you.

Jim MacDonald, **Francois Tetreault**, and **Octavian Stelescu**, for giving me an opportunity to test my host-based virtual machine analysis tool.

Bahman Khabiri, for always being a true friend.

The Dorsal lab, for its permissive atmosphere, stimulating lab-meetings with smart, hard-working colleagues and many laughter. **Suchakrapani Datt Sharma**, you were the perfect lab-partner to work with when I started in this group. Thank you for all your enthusiastic hard work, sanity, wisdom and courage. **Francis Giraldeau**, for dealing with me. **Mohamad Gebai**, for inspiring this work. **Genevieve Bastien** and **Naser Ezzati Jivan**, for being so positive and, for helpful discussions, not to forget, for all your enthusiasm in training. **Abderrahmane Benbachir**, for all the hard work, for your flexibility, and all the nice times we've shared analyzing trace. **Houssem Daoud**, for his useless but funny discussions and for never feeling things as impossible. **Matthew Khouzam**, for being so positive, and for being a good human being. Finally, thanks to the rest of the group: **Julien Desfossez**, **Thomas Bertauld**, **Cédric Biancheri**, **Didier Nadeau**, **Paul Margheritta**, **Iman Kohyarnejad**, **Ahmad Shahnejat**, **Majid Rezazadeh**, **Loïc Prieur-Drevon**, **Adel Belkhiri**, **Pierre Zins**, **Marie Martin**, **Irving Muller**, **Quentin Fournier**, **Pierre-Frederick Denys**, **Anas Balboul**, **Loïc Gelle**, **Guillaume Champagne**, and **Arnaud Fiorini**.

Finally and most importantly, I would like to deeply thank my loving **parents**, my dear **parents-in-law**, my beloved wife, **Mahsa**, my dearest son, **Ayden**, my dear brothers, **Hamed** and **Hesam**, and my dear sisters-in-law, **Mojdeh** and **Shirin**, for making me who I am as a human being. There are no words in any language that can express their importance in my life as a constant source of encouragement and inspiration.

RÉSUMÉ

L'infonuagique a beaucoup gagné en popularité car elle permet d'offrir des services à coût réduit, avec le modèle économique Pay-to-Use, un stockage illimité avec les systèmes de stockage distribué, et une grande puissance de calcul grâce à l'accès direct au matériel. La technologie de virtualisation permet de partager un serveur physique entre plusieurs environnements virtualisés isolés, en déployant une couche logicielle (Hyperviseur) au-dessus du matériel. En conséquence, les environnements isolés peuvent fonctionner avec des systèmes d'exploitation et des applications différentes, sans interférence mutuelle. La croissance du nombre d'utilisateurs des services infonuagiques et la démocratisation de la technologie de virtualisation présentent un nouveau défi pour les fournisseurs de services infonuagiques.

Fournir une bonne qualité de service et une haute disponibilité est une exigence principale pour l'infonuagique. La raison de la dégradation des performances d'une machine virtuelle peut être nombreuses. **a** Activité intense d'une application à l'intérieur de la machine virtuelle. **b** Conflits avec d'autres applications à l'intérieur de la machine même virtuelle. **c** Conflits avec d'autres machines virtuelles qui roulent sur la même machine physique. **d** Échecs de la plateforme infonuagique. Les deux premiers cas peuvent être gérés par le propriétaire de la machine virtuelle et les autres cas doivent être résolus par le fournisseur de l'infrastructure infonuagique. Ces infrastructures sont généralement très complexes et peuvent contenir différentes couches de virtualisation. Il est donc nécessaire d'avoir un outil d'analyse à faible surcoût pour détecter ces types de problèmes.

Dans cette thèse, nous présentons une méthode précise permettant de récupérer le flux d'exécution des environnements virtualisés à partir de la machine hôte, quel que soit le niveau de la virtualisation. Pour éviter des problèmes de sécurité, faciliter le déploiement et minimiser le surcoût, notre méthode limite la collecte de données au niveau de l'hyperviseur. Pour analyser le comportement des machines virtuelles, nous utilisons un outil de traçage léger appelé Linux Trace Toolkit Next Generation (LTTng) [1]. LTTng est capable d'effectuer un traçage à haut débit et à faible surcoût, grâce aux mécanismes de synchronisation sans verrous utilisés pour mettre à jour le contenu des tampons de traçage.

Notre méthode permet de détecter les différents états de vCPU, processus et fils d'exécution, non seulement à l'intérieur des machines virtuelles, mais également à l'intérieur des machines virtuelles imbriquées. Nous proposons également une technique d'analyse de chemin critique pour les machines virtuelles et les machines virtuelles imbriquées, permettant de suivre le chemin d'exécution d'un processus, à travers le réseau et les différentes couches de virtuali-

sation.

De plus, nous utilisons le traçage de l'hyperviseur pour extraire, d'une manière non intrusive, des données utiles permettant de caractériser le comportement des machines virtuelles à partir de l'hôte. En particulier, nous mesurons les périodes de blocage des machines virtuelles et le taux d'injection d'interruptions virtuelles, afin d'évaluer l'intensité d'utilisation des ressources. De plus, nous mesurons le taux de concurrences sur les ressources causées par l'hôte et les autres machines virtuelles, ainsi que les raisons de sorties du mode non privilégié vers le mode privilégié, révélant des informations utiles sur la nature de la charge de travail des machines virtuelles analysées. Ceci peut ensuite être utilisé pour optimiser le placement des machines virtuelles sur les noeuds physiques.

Nous avons implémenté nos méthodes d'analyse de machines virtuelles en tant que modules d'extension dans TraceCompass [2]. TraceCompass est un logiciel libre d'analyse des traces et des fichiers journaux. Il fournit un cadre d'applications permettant d'extraire des métriques et de générer des vues graphiques. Nous avons implémenté plusieurs vues graphiques pour TraceCompass. Une vue des processus affiche les processus des machines virtuelles (à n'importe quel niveau), leurs niveaux d'exécution du code et leurs états (en exécution et bloqués). Une vue de vCPU affiche les états des fils d'exécution vCPU, du point de vue de l'hôte. Elle représente les fils d'exécution vCPU avec les autres fils d'exécution de l'hôte. La vue du chemin critique affiche, de manière transparente, les chaînes de dépendances d'attente/réveil des processus des machines virtuelles, quel que soit leur niveau de virtualisation. Nos tests montrent que le surcoût de notre approche est d'environ 0.3%, car nos méthodes limitent la collecte de données au niveau de l'hôte.

ABSTRACT

Cloud computing has gained popularity as it offers services at lower cost, with Pay-per-Use model, unlimited storage, with distributed storage, and flexible computational power, with direct hardware access. Virtualization technology allows to share a physical server, between several isolated virtualized environments, by deploying an hypervisor layer on top of hardware. As a result, each isolated environment can run with its OS and application without mutual interference. With the growth of cloud usage, and the use of virtualization, performance understanding and debugging are becoming a serious challenge for Cloud providers. Offering a better QoS and high availability are expected to be salient features of cloud computing. Nonetheless, possible reasons behind performance degradation in VMs are numerous. **a)** Heavy load of an application inside the VM. **b)** Contention with other applications inside the VM. **c)** Contention with other co-located VMs. **d)** Cloud platform failures. The first two cases can be managed by the VM owner, while the other cases need to be solved by the infrastructure provider. One key requirement for such a complex environment, with different virtualization layers, is a precise low overhead analysis tool.

In this thesis, we present a host-based, precise method to recover the execution flow of virtualized environments, regardless of the level of nested virtualization. To avoid security issues, ease deployment and reduce execution overhead, our method limits its data collection to the hypervisor level. In order to analyse the behavior of each VM, we use a lightweight tracing tool called the Linux Trace Toolkit Next Generation (LTTng) [1]. LTTng is optimised for high throughput tracing with low overhead, thanks to its lock-free synchronization mechanisms used to update the trace buffer content.

Our proposed method can detect the different states of vCPUs, processes, and threads, not only inside the VMs but also inside nested VMs. We also propose a VM and Nested VM critical path analysis technique, to follow the execution path of arbitrary processes, over the network as well as through multiple virtualization layers.

Moreover, we propose host level hypervisor tracing as a non-intrusive means to extract useful features that can provide for fine grain characterization of VM behaviour. In particular, we extract VM blocking periods as well as virtual interrupt injection rates to detect multiple levels of resource intensiveness. In addition, we consider the resource contention rate, due to other VMs and the host, along with reasons for exit from non-root to root privileged mode, revealing useful information about the nature of the underlying VM workload. This can then be used for the efficient placement of VMs on hosts

We implemented our methods for analyzing VMs as separate modules in TraceCompass [2]. TraceCompass is an Open-source software for analyzing traces and logs. It provides an extensible framework to extract metrics, and to build views and graphs. We implemented several graphical views for TraceCompass. A VM process view shows the processes of VMs (in any level) with its level of code execution and states (all running and blocking states). A VM vCPU view displays the states for vCPU threads from, the host point of view. It represents vCPU threads along with other threads inside the host. The VM critical path view shows the extracted waiting / wake-up dependencies chains for the running processes, across VMs, for any level of virtualization, in a transparent manner. Our benchmarks show that the overhead for our approach is around 0.3%, because our methods limit their data collection to host level.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

NIST	National Institute of Standards and Technology
VM	Virtual Machine
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
AaaS	Application as a Service
vCPU	Virtual CPU
pCPU	Physical CPU
OS	operating system
VMX	Virtual Machine eXtensions
SVM	Secure Virtual Machine
VMM	Virtual Machine Monitor
KVM	Kernel based Virtual Machine
Qemu	Quick emulator
VMCS	Virtual Machine Control Structure
NPT	Nested Page Tables
EPT	Extended Page Table
CI	Continuous Integration
SLA	Service Level Agreement
PaU	Pay as Use
KPI	Key Performance Indicator
LTTng	Linux Trace Toolkit next generation
PMU	Performance Monitoring Unit
eBPF	Extended Berkeley Packet Filter
AMP	Application Performance Management
AI	Artificial Intelligence
PCMONS	Private Cloud Monitoring System
XDR	eXternal Data Representation
SVM	Support Vector Machine
TSRSVM	Training Set Refreshed SVM

CHAPTER 1 INTRODUCTION

The Cloud architecture relies on multiple layers of vitualization, which leads to sharing resources, and maintaining good security, isolation between tenants, elasticity and scalability. In this way, infrastructure providers can maximize their profit by leveraging their resources for delivering services to thousands of cloud users.

The classical approaches for monitoring and debugging VMs is to use existing toolkits for debugging and troubleshooting software systems, or adding printout functions to the source code. These approaches are not tailored for virtual machine, and may add a large overhead to the system. On the other hand, tracing can be an important resource for analysing virtual machines, providing very insightful approaches to be further developed. Tracing is used for analysis, validation, optimization, and understanding the behaviour of complex multi-layered software systems [5].

The main focus of this thesis is to facilitate the analysis of VMs, and to enhance the comprehension of application executions inside VMs. This includes the development of several agent-less algorithms, using existing tracepoints in the Linux Kernel-based Virtual Machine (KVM) module, along with new added tracepoints, in order to analyse VM processes in different virtualization layers, along with their communication with other processes, in a distributed manner.

1.1 Challenges of Virtualized Environment Analysis

While the cloud architecture offers many benefits, monitoring and analysing such a large-scale distributed system, in terms of debugging and troubleshooting, is a big challenge for infrastructure providers. The challenge becomes even more complicated when VMs can execute nested hypervisors (Nested VMs). The Infrastructure as a Service (IaaS) provider scheme gives the cloud user the ability to manage and use their own hypervisor as a VM. However, such cloud applications become even more complex.

The other challenge in analysing VMs is to trace each and every layer, and to correlate the information from the different layers. Moreover, it generates a lot of redundant data, incurring a large overhead, and in most cases the IaaS administrator cannot install a tracing agent in the client VM. Situations where a tracing agent cannot be installed include: **1)** Because of security concerns, the cloud provider does not have access to the VMs. **2)** The VM kernel is too old to install a tracing tool. **3)** The VM kernel is closed-source and there is

no tracer available. **4)** The VMs is limited in terms of resources. Nonetheless, reducing the complexity of analyzing and monitoring VMs remains an important issue. The administrator may have to deal with thousands of VMs, and it is difficult to understand the root cause of issues.

As mentioned earlier, in the cloud environment, each VM and its applications have the illusion of having the whole system resources available to themselves, while in reality the ressources are aggressively shared in order to maximize efficiency and profit. Correspondingly, an application could be tested and validated in a non-virtualized environment but fail to run in a virtualized environment. Thus, it is important to quantify the resource usage of an application while its behaviour may change over time. Tracking these changes helps IaaS providers in tuning the configuration for each VM and host. Of the many different VM analysis techniques proposed in the literature, only a few support agent-less operation (e.g, Virtual Machine Introspection (VMI) techniques [6]).

In summary, for such complex virtualised distributed environments, there is a need to elaborate more sophisticated algorithms and techniques for performance analysis.

Research Questions: The state-of-the art VM analysis techniques have not addressed the challenges of efficiently analysing VMs in different layers. In view of the issues discussed earlier, we define four important research questions that have not been addressed in-depth in this domain:

- How to detect deviations in virtual machine performance, when a multi-layer tracer may add a significant load?
- What are the most relevant features for detecting over-utilized and under-utilized physical hosts?
- How to track the application changes and quantify its resources needs?
- How to cluster VMs based on workload characteristics? What are the most relevant features to classify a VM?

1.2 Research Objectives

To obtain a highly detailed view of a VM, while keeping the tracing overhead almost negligible, an agent-less technique is proposed, to avoid the cost of tracing for each and every VM and layer. In addition, there is a need to elaborate more automated techniques for VM performance analysis.

For this research, the general objective is to solve the problem of analysing multi-layered and distributed virtualized environments.

Furthermore, we identified specific objectives of our research as follows :

- To develop a vCPU and process state detection algorithm that can automatically investigate the root cause of latencies in VMs.
- To study the behaviour of VMs in any level of virtualization.
- To develop a new algorithm for dependency analysis of VM processes.
- To propose an algorithm to reduce the complexity of analysing and monitoring VMs by leveraging the VM clustering technique.

1.3 Contributions

In line with the research objectives stated above, this thesis presents the following original contributions in the field of virtual machine analysis:

- A fine-grained VM vCPU and nested VM vCPU state analysis (including reasons for blocking).
- An approach to follow execution paths of arbitrary processes, over the network as well as through multiple virtualization layers.
- A clustering method to characterize the workload of VMs.
- Several graphical views to present a time-line for each process and vCPU, with different states.

1.4 Outline

The chapters are organized as follows. Section 2 presents some background information and a summary of other existing approaches for VM analysis, nested VMs, feature extraction, and VM clustering. Chapter 3 outlines our research methodology and explains the process of generating research leads, identifying problems, actionable items, specific milestones and eventual outcomes, in terms of research papers. It presents an overall view of the body of this research. This is followed by three journal publications (research papers), presented in Chapters 4, 5, and 6.

In Chapter 4, we first present a new host hypervisor based analysis method which can investigate the performance of VMs in any nesting level. We also propose a vCPU state builder algorithm, and Nested VM state detection algorithm, to detect the state of vCPUs and nested VMs along with the reason for being in that state. Then, we propose a new approach for profiling threads inside the VMs by host tracing. This article is titled "virtFlow: Guest Independent Execution Flow Analysis Across Virtualized Environments" and appeared in the IEEE Transactions on Cloud Computing (TCC).

The second article, in Chapter 5, discusses a new execution-graph construction algorithm to extract the waiting / wake-up dependency chains from the running processes across VMs, for any level of virtualization, in a transparent manner. This article is titled "Critical Path Analysis through Hierarchical Distributed Virtualized Environments using Host Kernel Tracing" and was submitted to the IEEE Transactions on Cloud Computing (TCC).

The third article, in Chapter 6, presents a novel host level hypervisor tracing technique as a non-intrusive way to extract useful features, enabling a fine grained characterization of VM behaviour. In particular, we extracted VM blocking periods as well as virtual interrupt injection rates to detect multiple levels of resource usage intensity. This article is titled " Host-based Virtual Machine Workload Characterization using Hypervisor Trace Mining" and was submitted to Journal of ACM Transactions on Modeling and Performance Evaluation of Computing Systems.

Finally, in Chapters 7 and 8, we present a summary and discussion of our research contributions and their impact on the tracing ecosystem, and we provide recommendations for future work in this field.

1.5 Publications

The chapters outlined above are based on the published/submitted papers mentioned in this section. The publications are divided into two sections: journal papers included in this thesis, and conference papers.

1.5.1 Journal Papers

1. H. Nemati and M. R. Dagenais, "virtFlow: Guest Independent Execution Flow Analysis Across Virtualized Environments," in IEEE Transactions on Cloud Computing. doi: 10.1109/TCC.2018.2828846

2. Hani Nemati, Francois Tetreault, Jason Puncher, and Michel Dagenais, "Critical Path

Analysis through Hierarchical Distributed Virtualized Environments using Host Kernel Tracing," submitted to IEEE Transactions on Cloud Computing.

3. Hani Nemati, Seyed Vahid Azhari, Mahsa Shakeri, and Michel R. Dagenais, "Host-based Virtual Machine Workload Characterization using Hypervisor Trace Mining," submitted to ACM Transactions on Modeling and Performance Evaluation of Computing Systems.

1.5.2 Conference Papers

1. H. Nemati and M. R. Dagenais, "VM processes state detection by hypervisor tracing," 2018 Annual IEEE International Systems Conference (SysCon), Vancouver, BC, 2018, pp. 1-8. doi: 10.1109/SYSCON.2018.8369612

2. H. Nemati, G. Bastien and M. R. Dagenais, "Wait analysis of virtual machines using host kernel tracing," 2018 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, 2018, pp. 1-6. doi: 10.1109/ICCE.2018.8510984

3. Hani Nemati, Suchakrapani Datt Sharma, and Michel R. Dagenais. 2017. Fine-grained Nested Virtual Machine Performance Analysis Through First Level Hypervisor Tracing. In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17). IEEE Press, Piscataway, NJ, USA, 84-89. DOI: https://doi.org/10.1109/CCGRID.2017.20

4. S. D. Sharma, H. Nemati, G. Bastien and M. Dagenais, "Low Overhead Hardware-Assisted Virtual Machine Analysis and Profiling," 2016 IEEE Globecom Workshops (GC Wkshps), Washington, DC, 2016, pp. 1-6. doi: 10.1109/GLOCOMW.2016.7848953

5. H. Nemati and M. R. Dagenais, "Virtual CPU State Detection and Execution Flow Analysis by Host Tracing," 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom), Atlanta, GA, 2016, pp. 7-14. doi: 10.1109/BDCloud-SocialCom-SustainCom.2016.13

6. Hani Nemati, Seyed Vahid Azhari, and M. R. Dagenais "Host Hypervisor Trace Mining for Virtual Machine Workload Characterization," IEEE International Conference on Cloud Engineering (IC2E), Prague, 2019.

CHAPTER 2 LITERATURE REVIEW

Cloud computing became a recent revolutionary technology because it reduces the companies IT costs by cutting down both capital and operating costs using resource sharing. It also offers scalability to the businesses by scaling up or scaling down its existing resources to accommodate business changes and requirements. Unfortunately, identifying and troubleshooting VM performance degradations, capacity requirements, and configuration issues can be challenging and time-consuming, as performance issues can originate from any number of root causes. This survey explores interesting techniques for analyzing, monitoring, debugging, and troubleshooting virtual environments.

This chapter delves into available strategies, tools and methods for analysis, troubleshooting and debugging virtual machines. First, we categorize the existing techniques into two major classes: 1) agent-based techniques, and 2) agent-less techniques. In agent-based techniques, an agent (small executable) will be installed on VMs to monitor the service. On the other hand, agent-less techniques limit their data collection to the physical host level, without internal access to the VMs. Both agent-based and agent-less monitoring techniques have advantages and disadvantages which will be covered in different subsections. We further categorize the existing techniques based on the technology behind data collection (e,g,. Logging, tracing, etc.).

The rest of this chapter is organized as follows: the first section presents some common definitions and terminology, in the world of virtual machines, along with background information about virtualization technology, cloud architectures, nested VMs, and monitoring system components. Follows an overview of interesting tracers and tools. Finally, subsection 2.5 presents the state of the art for commercial and academic analyzers for VMs.

2.1 Background Information

Our aim in this section is to describe some common definitions and terminology in the world of virtual machine analysis. This section is essential for understanding the rest of the survey.

2.1.1 Cloud Computing

Cloud Computing refers to a service model in which the services can be delivered to users, irrespective of how or where. It is a parallel and distributed system that uses virtual computing resources to provide services for users based on a service level agreement. We will elaborate more on virtualization technology in subsection 2.1.4. The virtual computing resource pool can be reconfigured according to the traffic load and even the users location. Cloud Computing is a combination of several concepts like Internet delivery, Pay-per-Use-On-Demand utility computing, virtualization, grid computing, distributed computing, storage elasticity, content outsourcing, and Web 2.0. Based on the definition of Cloud Computing by the U.S. National Institute of Standards and Technology (NIST), a cloud model is composed of five essential characteristics, three service models, and four deployment models [7].

2.1.2 Cloud Service Models

Cloud computing can be divided into three layers, based on the definition of cloud computing by NIST [7]. These service models will be examined in the subsequent subsections.

Infrastructure as a Service (IaaS): The Cloud Provider offers fundamental computing resources (e.g., processor, storage and networks) where the customer has the ability to create and run arbitrary operating systems and applications. The customer is not a manager of physical resources but has control over operating systems, storage, and deployed applications. IaaS can attract enterprise and individual users by offering different billing methods and services to meet the needs of customers. Some powerful IaaS companies are Amazon Web Services, Bluelock, SCS and IBM [8].

Platform as a Service (PaaS): PaaS is a complete virtual platform, onto which developers can login from different terminals. The developers use pre-installed programming languages, libraries, and tools to develop their applications (e.g. custom cloud hosting apps). The PaaS users do not have access to physical resources, and the underlying infrastructure. Examples of PaaS providers are OpenShift, Microsoft Azure, Google App engine and IBM Smart Cloud [9].

Software as a Service (SaaS): SaaS is an application driven model that enables on-demand availability of software over the internet. It can provide many different applications in the cloud; the underlying cloud infrastructure is transparent to the customers. SaaS is also called Application as a Service (AaaS). Examples of SaaS providers include CVM solutions, Gageln, Knowledge tree and LiveOps [10].

2.1.3 Essential characteristics of Cloud Computing

Five essential characteristics of cloud computing are described below [11] [12] [13]:

On-demand self-service: The customer can select and change the computing capabilities without requiring human intervention.

Broad network access: Users can access a cloud system over the network, through standard mechanisms from any physical platform.

Resource pooling: The computing resources are assigned to customers using a multi-tenant model. The users are not aware of the resource location but may have the ability to specify the location at a higher level of abstraction (e.g., state or country).

Rapid elasticity: Capabilities should be able to scale outward and inward, based on traffic load, needed QoS and SLA.

Measured Service: The cloud system should be equipped with an analysis tool to automatically control and optimize resource usage. Furthermore, the analysis tool should send reports to both the provider and consumer of the service.

2.1.4 Virtualization Technology

The concept of virtualization was introduced by IBM in early 1970s, when they were working on developing time-sharing solutions for the mainframes [14]. The basic idea of time-sharing is sharing computing resources among many users by applying multitasking. Multiple tasks, also known as processes, share common processing resources such as a CPU, aiming to increase the utilization of computers. By introducing this technology, the illusion of parallelism is achieved and the cost (time and money) of using a computer is reduced significantly. The advantages behind the time-sharing technology were sufficient for driving the development of virtualization in industry. Nowadays, servers are equipped with powerful processors, high capacity hard drives, a large amount of memory and I/Os, that may often remain idle. The best way to increase resource utilization is using virtualization. Virtualization enables the

physical machine to run one or more virtual machines at same time. A virtual machine (VM) is a software implementation of a machine (e.g., a computer) that executes programs, just like a physical machine [15].

Modern Intel x86 (Intel-VT) and AMD CPU cores support a specific VM instruction set in order to allow multiple operating systems (OSs) to share the physical CPU in a safe and efficient manner. Intel named its VM instruction set as Virtual Machine eXtensions (VMX) while AMD called it the Secure Virtual Machine (SVM). There are two VMX operating modes: VMX root and VMX non-root. In VMX root operation, the Virtual Machine Monitor (VMM) or Hypervisor is executed. In the VMX non-root mode, the guest software is run.

An Hypervisor is an essential component of the virtualization stack, that creates a platform to run a pool of VMs on top of a physical machine. Hypervisors are classified as native and hosted [16]. The native hypervisor is an operating system that runs directly on a physical machine to control and monitor the VMs. The hosted hypervisor is designed to run on top of another operating system. It is developed to add a distinct software layer on top of the operating system. For example, Xen and VMWare ESX are native hypervisors, and VirtualBox and VMWare workstation are hosted hypervisors. Figure 2.1 depicts the architecture of native VMs on top of a native hypervisor.

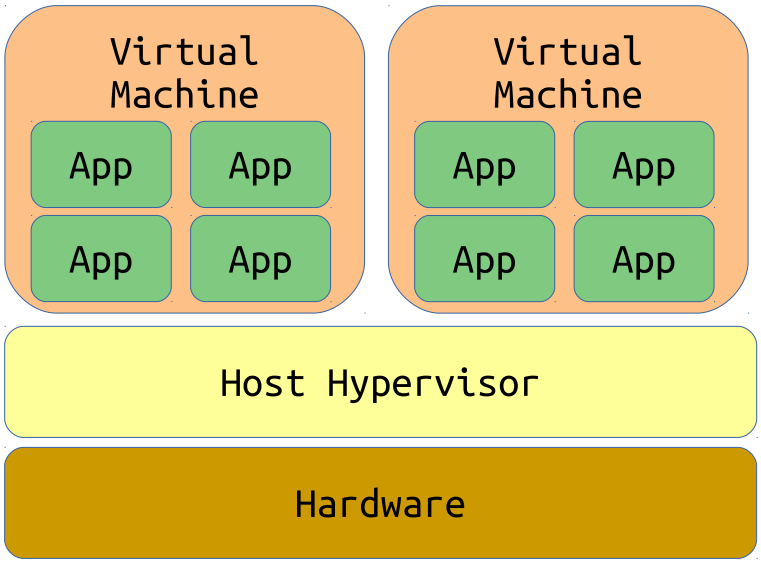


Figure 2.1 Architecture of a native hypervisor

A hypervisor is a VM manager which creates and manages the VMs. VMs that run on top of native hypervisors may perform better, since the VM code executes directly on a physical CPU (pCPU). On the other hand, an ideal hypervisor is just an accelerator focusing on the CPU. Therefore, another component is needed to manage other hardware such as I/O

peripherals, which is achieved with an userspace application. For example, in Linux, the Kernel based Virtual Machine (KVM) acts as a native hypervisor to run VMs, and Qemu acts as an userspace application to emulate other hardware. Figure 2.2 shows the architecture of a Qemu/KVM VM.

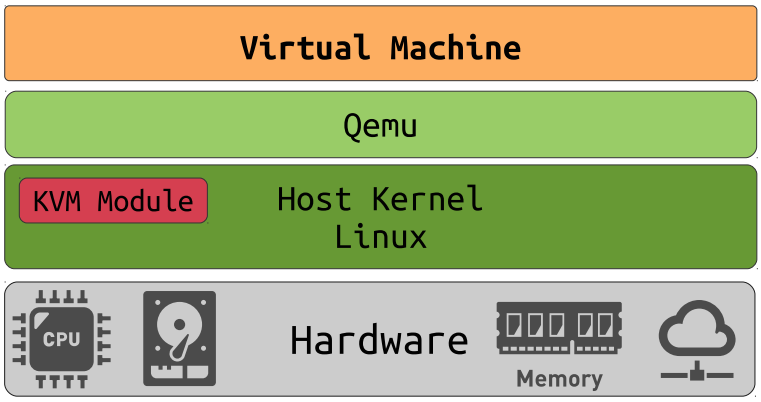


Figure 2.2 Architecture of Qemu/KVM

2.1.5 VM and Nested VM instruction set

As mentioned in subsection 2.1.4, Intel and AMD support special instruction sets for their virtualization technology. In this subsection, we elaborate more on VMX, the Virtual-Machine Control Structure (VMCS) and interrupt handling for VMs and Nested VMs.

VMX operations are divided into two main categories. In the VMX root mode, the hypervisor is executed and in the VMX non-root more, the guest software is run. Correspondingly, there are two kinds of transitions. VM Entry is the transition from VMX root mode to VMX non-root mode (from hypervisor mode to guest mode). In the guest mode, non-privileged instructions of VMs are executed as non-root mode. Executing privileged instructions in guest mode causes an exit to the hypervisor mode. VMX Exit is the transition from VMX non-root mode to VMX root mode. The hypervisor handles the privileged instruction and the enters to guest mode. In each transition, the environment specifications of the VMs and hypervisor are stored in a structure in memory named Virtual Machine Control Structure (VMCS) [17]. The hypervisor stores and retrieves the VM information using the `VMREAD`, `VMWRITE`, and `VMCLEAR` instructions [17].

The challenge of monitoring VMs increases significantly when VMs may run a nested hypervisor (Nested VM). A nested VM is a guest that runs inside another VM, on top of two or more hypevisors [18]. The IaaS provider scheme enables the cloud user to manage and use

his own hypervisor as VM. However, the diagnosis of any latency or response time problem in Nested VMs is quite complex, due to the different levels of code execution.

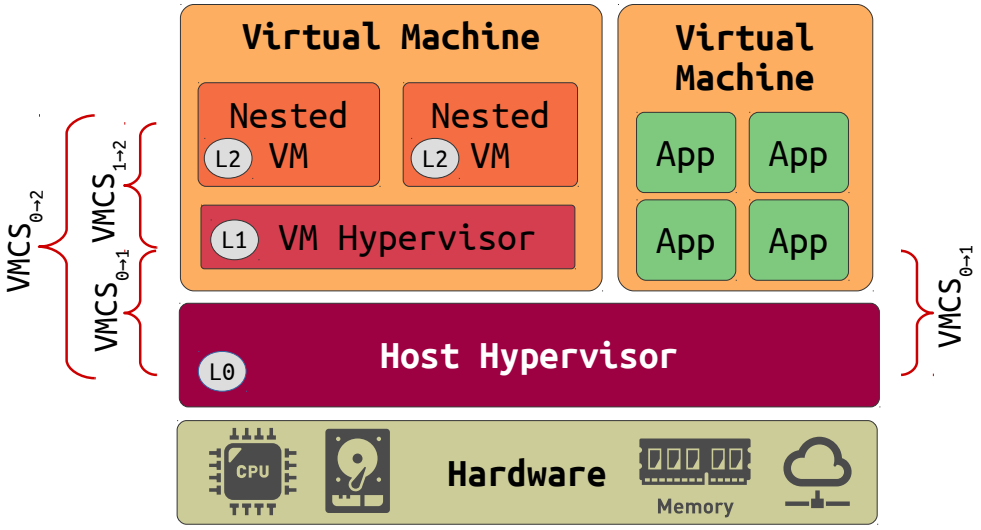


Figure 2.3 One-Level Nested VM Architecture for VMX

Nested VMs add new challenges for CPU, I/O and memory virtualization. New processors from Intel and AMD support Nested VMs. Figure 4.2 shows a single-level architecture for nested VMs. In a single-level architecture, executing any privileged instruction, from any level of nested VMs, returns to the host hypervisor (L_0). In this case, the VM hypervisor (L_1) has the illusion of running the code of the nested VM (L_2) directly on the physical CPU. However, privileged instructions of nested VMs should be handled by the highest privileged level. Since L_1 is not the highest privileged level, L_0 handles it. As a result, whenever any hypervisor level or VM executes privileged instructions, the L_0 trap handler is executed. This VMX emulation can go to any level of nesting.

Usually, there is one VMCS for each vCPU ($VMCS_{01}$). However, for one level of nested VMs, there are three VMCSes per vCPU. The VM hypervisor uses $VMCS_{12}$ to contain the environment specifications of nested VMs. As we mentioned before, the code of nested VMs can be executed directly on the host hypervisor. In this case, the host hypervisor prepares $VMCS_{02}$ to save and to store the state of nested VMs at each VM exit and VM entry. Moreover, the host hypervisor creates $VMCS_{01}$ to execute the code of the VM hypervisor. From the host perspective, $VMCS_{12}$ is not valid, but the host hypervisor benefits from that to update some fields in $VMCS_{02}$ for each VMX transition.

AMD and Intel implement additional hardware support for virtualized MMUs in normal VMs called Nested Page Tables (NPT) and Extended Page Table (EPT), respectively. For memory

virtualization, nested VMs add another level of address translation that is not implemented in hardware. KVM and other existing hypervisors handle Nested VM memory management in software.

For I/O virtualization, Nested VMs can access I/O directly or by emulation. In the case direct access, it has near-native performance. With emulation, the performance is not as good as direct access, but any I/O device can be emulated in nested VMs [19].

Contemporary use of nested VMs is more for the purpose of software scaling, compatibility, and security. In addition, many network services can be virtualized (a main goal of NFV) and hosted on nested VMs. Software as a Service (SaaS) providers are the best clients of nested virtualization. SaaS providers can encapsulate their software in a nested VM on an existing cloud infrastructure (e.g, Google Cloud and Amazon AWS). Ravello [20] [21] has implemented a high performance nested virtualization called as HVX. It allows the user to run unmodified nested VMs on Google cloud and Amazon AWS, without any change.

Nested virtualization is also used for Continuous Integration (CI). CI integrates code, builds modules and runs tests when they are added to the larger code base. Because of security and compatibility issues, the building and testing phases should be run in an isolated environment. CI service providers can execute each change immediately in a nested VM.

McAfee Deep Defender [22] is another example of nested VM use. For security reasons, it has its own VMM. CloudVisor [**?**] also propose a method to protect VMs using nested virtualization. This tiny nested VM provides security monitoring and protection to the hosted VMs. Furthermore, one of the features in Windows 7, in the professional and ultimate editions, is the XP mode [23]. In this mode, a VM runs Windows XP for compatibility reasons. Thus, Windows 7 users can execute Windows XP applications without any change. Correspondingly, the XP mode will be run in a nested VM if Windows 7 is running in a VM.

2.2 Cloud Analysis tools: The Need

A mentioned in subsection 2.1.3, cloud computing has several essential characteristics. In order insure that these characteristics are being satisfied, an analysis tool is necessary. Here are some of the important requirements to analyze VMs:

Failure Detection: Fault in cloud computing may happen in many ways, such as a natural disaster, a hardware problem, or a software bug. One of the main goals in cloud computing is to enable reliable and resilient services. Migration of VM instances is the key to have a well balanced and failure free environment. In this case an analysis tool should predict the failure of a VM instance and prepare a migration destination. As a result, by continuously analyzing and detecting failure causes, a cloud provider could offer a failure-free environment. Another approach is to have a self-healing mechanism to detect the root cause of a problem in the system and fix it. All of these approaches need a proper analysis tool.

Resource Management: In a single domain and single user system, a designer could evaluate workload bottlenecks and identify overload situations. Since the Cloud environment consists of thousands of VM instances, sharing resources, resource management is more sophisticated. By contrast, the workload of VM instances can vary in many situations. Therefore, cloud providers need to use a resource management system that continuously keeps track of resource status, in different virtualization layers. The resource management system should monitor, analyze, and predict future resource demands for each VM instance.

Application Design: Writing cloud-friendly applications is an important aspect that should be considered when the application runs in a virtualized environment. Applications that execute many privileged instructions can reduce the performance and cause a high overhead. Programmers can examine their application, using different tracing techniques, to analyze the performance application executions.

Cost reduction: Sharing resources plays an important role in efficient resource usage and reducing operational costs. Moreover, the most important motivation of infrastructure providers is maximizing profit by leveraging their resources for delivering service to thousands of cloud users. In order to maximizing profit, cloud providers uses an analysis tool to measure and forecast the sufficient level of resources required for each VM instance.

Accounting and Billing: Cloud providers charge customers following a model called Pay as Use (PaU). In this case, it is important to monitor the resource usage of each instance for accounting and billing. A monitoring tool is important for both the customer and cloud provider, since in case of SLA violations, the cloud provider should pay back money as a penalty.

SLA Management: Since cloud computing offers a Pay as Use model, the customers may define some expectations for the services delivered by cloud providers. A Service Level Agreement (SLA) is a well-defined agreement between the user and provider. If the SLA with a customer is violated, there is the possibility of a penalty and customer dissatisfaction. There are different strategies being used to reduce SLA violations, like SLA-based scheduling policies, and SLA-based resource allocators. In all these strategies, the cloud provider needs to implement a cloud analysis tool to measure different Key Performance Indicators (KPI). This analysis tool needs to keep track of each defined SLA clause precisely.

2.3 Cloud Analysis tool: Properties

Many tracing and analysis tools have been used for monitoring the performance of processes, the OS, and whole systems. Most of these tools deal with a single system in a single domain. By introducing multi-domain and multi-layer distributed systems, these tools need to evolve to capture the new systems characteristics. In order to satisfy all the requirements of a system, it is necessary to analyze the system from different angles. In this subsection, some important capabilities and features of a suitable analysis tool are described as follows:

Available: It is important to insure that the analysis system itself is fully functional. This may be achieved by checking the result of system analysis. Log aggregation tools may also be used to trace across multiple systems [24].

Open-source: the cloud infrastructure and cloud software management are continuously changing and expanding. Therefore, the analysis tool needs to be adaptable. There are presently numerous cloud analysis tools which are closed-source, and information about how they monitor virtual hardware is a secret. The analysis tools in this category cannot adapt themselves at the rapid pace of technological changes, and cover a smaller scope of features and abilities [25]. In contrast, opensource analysis tools are extensible. Anyone can inspect, modify, and enhance them [26].

Resolution: Sometimes, to find some serious faults in a cloud setup, the analysis tool may need to provide high resolution results. For instance, in such a high resolution analysis tool, each event should have nanosecond precision timestamps.

Multi-Layer: The cloud is a multi-layered distributed system that can be modelled in 6 different layers: Guest Application, Guest Operating System, user-space VMM, kernel-space VMM, Host Operating System, Host Hardware. Analyzing each layer individually may not reveal important information about problems of cloud tenants. The cloud analysis tool should be able to trace and analyze most layers at the same time.

Scalable: Nodes in a cloud environment are distributed within a wide area. The analysis tool should be scalable and distributed to gather data from each node and each host.

Dynamic: An analysis tool should be dynamic in the sense that it deals with a changing virtual environment. Static analysis tools cope with a specific system, where all the

statically inserted probes gather information from the system. Dynamic analysis tools may add probes and can filter events at run-time.

Accurate: Accuracy is important for the cloud environment, since cloud computing offers to the end user the ability of accessing a pool of resources with the PaU model. To provide better QoS, an SLA is agreed upon by both the cloud provider and the customer, and the cloud provider has to pay penalties, in case of SLA violations. Therefore, it is vital for the analysis tool to provide accurate information. The Analysis tool could gather sampled data. To collect relevant data, there is trade-off between accuracy and sampling frequency. The sampling frequency could be as small as each event, in which case it becomes event-based data. We elaborate more on sampling-based and event-based data gathering in subsection 2.4.1

Adaptive: Cloud computing offers a variety of services to the end-user. Given the large number of customers with different levels of satisfaction, the analysis tool should adapt itself to each service configuration and provide suitable reports .

Portable: An analysis tool should perform in any environment regardless of its platform. It is necessary since the analysis tool copes with wide penetration across federated cloud.

Robust: It is important for an analysis tool to cope with errors and new situations while continuing its operation. This is more critical since it should monitor a multi-tenant environment where a fault in one virtual environment could have a devastating effect on the whole system.

Overhead: The analysis tool should add a low overhead. For instance, if there are one thousand nodes and if the analysis tool adds just 1% overhead, the cloud provider needs 10 extra CPUs to handle the data gathering load for the analysis tool.

2.4 Monitoring System Architecture

A monitor is a system that observes the behaviour of a target system to reach a decision about correctness. Figure 2.4 represents different components of a monitoring system. It has five different components: Aggregator, Analysis component, Visualization component, Alert and Tuning system. The data provider represents the technical means to explore the system state and its changes. The aggragator component gathers data and sends it to the analysis component. The analysis component processes the data and extracts meaningful information to represent the performance of a target system. The analyzed data is sent to the visualization component. Then, the visualization component generates several views for system administrators and clients. It also could trigger an alert and send it to the tuning component to enforce an action. In the following subsection, we elaborate more on the technology behind the data provider.

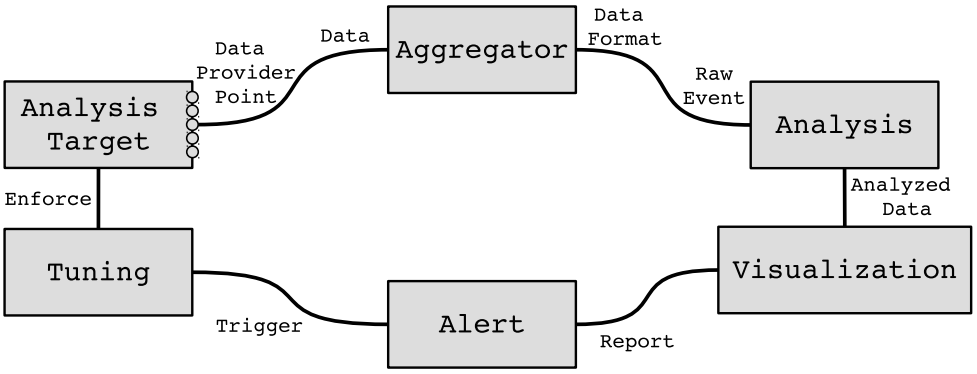


Figure 2.4 A monitoring system and its components

2.4.1 Technology behind data providers

As Figure 2.4 shows, the first step to monitor a system is the data provider. Time-based and Event-based measurements are two techniques for collecting data. In Time-base measurements, all the metrics are captured at a constant interval. Granularity and quality of metrics collection are dependent on time intervals. Longer time intervals might not be acceptable to reveal errors and problems, but add less overhead. In contrast, using short time intervals adds more overhead. The advantage of Time-based measurements is that the sample rate can be controlled to adjust the overhead. In Event-based measurements, an event with time-stamp is generated when the tracer encounters an enabled event. This method gives the highest granularity and quality for collecting metrics. Furthermore, modern tracers require little overhead and thus can be used in production environments. In this subsection, different

technologies behind data providers are studied.

2.4.1.1 Logging

In computing, instrumentation is a code section that is added to any part of a target system to gather information about the execution. A log entry is a message from the Operating System or application code to indicate the occurrence of an event. A log entry has a body, content and time, and is written to a logfile. It is used for statistical analysis and reporting. Logging systems are not efficient for high frequency events, and it is better to use other kinds of data providers (e,g,. tracing). For example, Amazon Web Service (AWS) uses coarse-grained log monitoring techniques [27] for the horizontal scaling of its instances.

2.4.1.2 Debugging

Debugging is mainly used in software development for locating and correcting code errors. It is also part of the software testing process. One of the most common program debugging technique is `printf` to display the value of variables. GDB is a common debugger in Linux [28], which uses the `ptrace` system call to interact with the process being debugged.

2.4.1.3 Tracing

Tracing is a very fast system-wide logging mechanism. Tracing usually involves inserting a special instruction or sequence of instructions (instrumenting), named a tracepoint, in the code. Instrumentation can happen either at run time or compile time. Instrumentation at run time is called dynamic binary instrumentation, and instrumentation at compile time is called static instrumentation. In the case of static instrumentation, there is an extra recompilation time and the need for restarting the application [29]. It also needs the source code of the application. In contrast, for dynamic instrumentation, code is injected without the knowledge of the application source code at run-time [30]. The tracepoint may simply call a user-defined hook, or call a function that is part of the kernel tracing infrastructure. Tracing can be divided into hardware and software traces, based on the origin of the event. In modern CPUs, there is the capability of recording a complete trace of the instructions executed. For example, Intel Processor Trace (Intel-PT) traces every branch, in order to record enough information to reconstruct the whole sequence of instructions executed. After recording the trace, it merges the recorded trace with the program binary to generate the execution flow of the program execution. As a result, it does not need to record each and every instruction execution [31]. In software tracing, all the tracepoints are in the program

code, and no special hardware is used to generate trace. Tracepoints are divided into static and dynamic instrumentation, based on how they are added to the code. In order to add a static tracepoint, the source code of program is needed and a recompilation is required. With dynamic instrumentation, the tracepoint is added directly into a running process, possibly using the debugging information generated at compilation time.

Tracing can also be divided into user-space and kernel tracing. In user-space tracing, the events are collected from user-space (the application code). In kernel tracing, events are generated by the operating system executable, the user application does not need to be instrumented. For example, Linux uses the `TRACE_EVENT` macro to provide static tracepoints. The tracer can hook a function to these tracepoints and collect events. It provides information about the operating system internals (e.g. thread scheduling, interrupts) and all the interactions between processes and the operating system (e.g. system calls) but little information about the internal logic of applications.

2.4.1.4 Sampling

Sampling consists in interrupting an unmodified application to sample some value (e.g. current executing address, current stack, number of allocated bytes). The sample rate should be controlled since it determines the overhead and accuracy trade-off. Most modern CPUs provide embedded performance counters like the number of cache faults at each cache level, the number of branch stalls and a large number of other metrics. OProfile [32] and Perf [33] are some performance analysis tools which use performance counter sampling.

While sampling-based tools are very useful to measure the average performance with a low overhead, event-based tools like tracers are more interesting for debugging specific problems. Indeed, you can follow the detailed trace of events leading to the problem. Therefore, being interested in debugging problems in layered cloud architectures, we focus here on tracing tools which are studied in the next subsection.

2.4.2 The aggegator component: Tracing and Profiling Tools

In this subsection, we study different available tracing tools. Tracing tools provide different techniques to aggregate data from the target system. Moreover, some tracing tools implement basic analyses to send to the visualization component.

Table 2.1 Available Kernel Tracepoints and their type

Types	Family	Events
Virtual Machine	Processor	kvm_set_irq, kvm_ioapic_set_irq,kvm_msi_set_irq, kvm_ack_irq, kvm_mmio,kvm_fpu, kvm_entry, kvm_exit,kvm_hypercall, kvm_hv_hypercall, kvm_pi, kvm_cpuid , kvm_apic,kvm_inj_*, kvm_apic_accept_irq, kvm_msr,kvm_pic_set_irq, kvm_apic_ipi„, kvm_eoi, kvm_pv_eoi,kvm_nested_*, kvm_emulate_insn„kvm_write_tsc_offset, kvm_update_master_clock,kvm_track_ts , kvm_apic_accept_irq
	Memory	kvm_age_page, kvm_try_async_get_page, kvm_async_*, kvm_page_fault, kvm_mmu_*
Host	OS	module_*, printk_console, random_*, rcu_utilization, regulator_*, gpio_*, sched_*, signal_*, workqueue_*
	Processor	power_*, irq_*, timer_*
	Memory	kmem_*, mm_*
	Disk	block_*, jbd2_*, scsi_*
	Network	napi_poll, net_*, skb_*, sock_*, rpc_*, udp_fail_queue_rcv_skb
	Graphics	asoc_snd_*, v4l2

2.4.2.1 LTTng

The Linux Trace Toolkit next generation (LTTng) is designed to offer very low overhead kernel and user-space tracing [34]. It implements a very fast lock-free, wait-free read-copy-update (RCU) buffer to store data and eventually copy it on disk. A very low overhead, which remains almost constant when the number of parallel cores increases, because of per-core buffers, makes LTTng the default tool for tracing real-time application in Linux. LTTng supports both static and dynamic tracing. It can add dynamic tracepoints with Kprobe. In addition, static tracepoints can be added in the source code of the kernel, as well as in user-space programs with UST. LTTng also provides hooks to the `TRACE_EVENT` macro for kernel events. LTTng exports the events in the Common Trace Format (CTF) to be written on disk [35].

Figure 2.5 shows the different components in LTTng and how they interact with application and the Linux kernel. The most important component is the session daemon. It is responsible for managing and controlling the other components. The LTTng kernel modules include: a set of probes to be attached to Linux kernel tracepoints and to the entry and exit of system calls. The Ring buffer module is an implementation of a ring buffer where the consumer daemon reads events. The LTTng Consumer daemon shares the ring buffer between user-space and

kernel-space to collect kernel trace data and copy it to disk. In [36], the authors provide a map between kernel tracepoints and their system activity on the host. The extended version of mapping between kernel tracepoints and their types, for both host and kernel, is depicted in Table 2.1

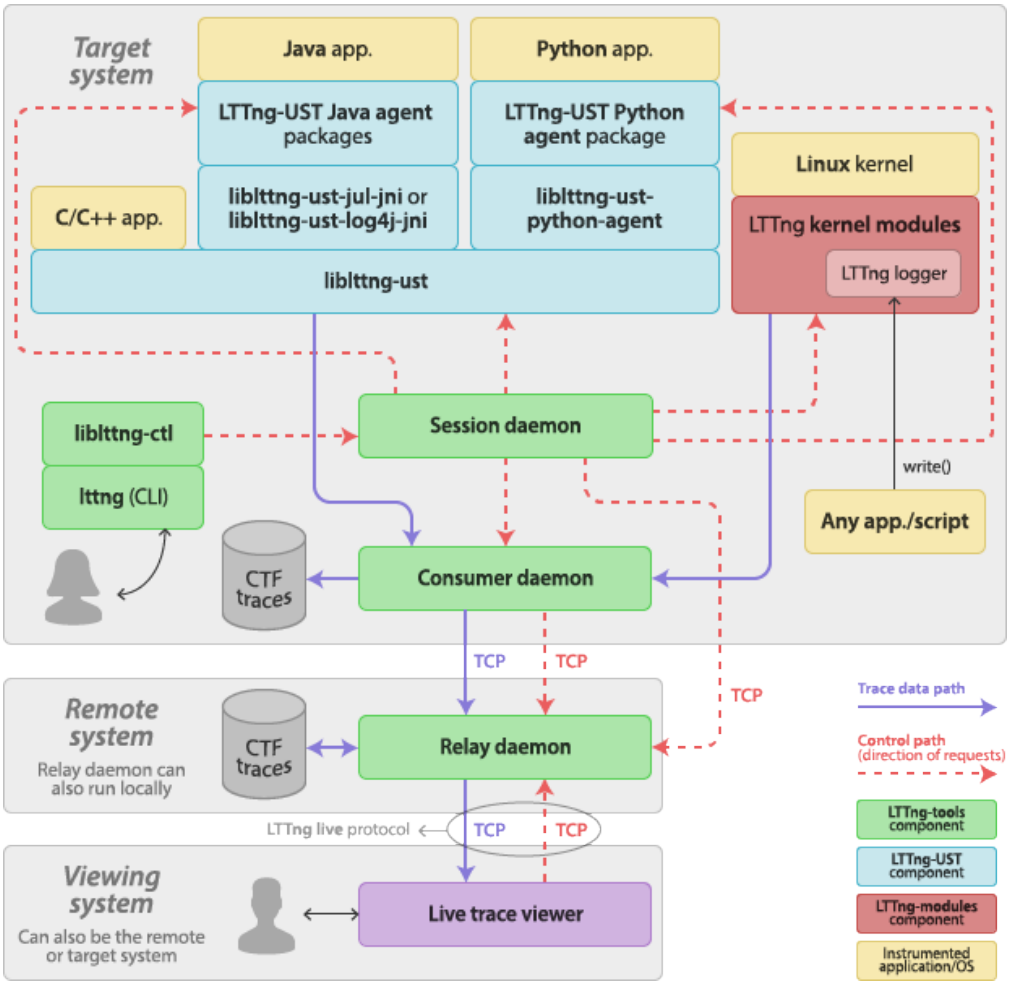


Figure 2.5 LTTng components and their tracing path [3]

LTTng is currently the fastest kernel/user-space tracing toolkit for Linux [37]. The major factor for such improvement is its lock-free implementation of the LTTng ring buffer.

2.4.2.2 Perf

Perf is a monitoring tool inside the Linux kernel mostly used for sampling and profiling. Perf can access and gather data from the hardware Performance Monitoring Unit (PMU), like cache misses in different levels, miss branch predictions, TLB hits, and so on. Perf can hook to `TRACE_EVENT` macros in the kernel and save events at run-time in the perf ring-buffer.

2.4.2.3 Ftrace

Ftrace is an older tracer included in the Linux kernel that allows function tracing, system calls tracing, dynamic instrumentation and so on. It provide function graphs that show the entry and exit of all functions at the kernel level. It supports Kprobe for dynamic instrumentation and provides hooks to the `TRACE_EVENT` infrastructure. It keeps the events in memory and does not write them to disk automatically. In Ftrace, the size of a payload is limited to the size of a page. The ring buffer is implemented as a linked list and a buffer page can be read once it is full.

2.4.2.4 SystemTap

SystemTap is a tracing tool which allows inserting tracepoints dynamically, as well as collecting events from tracepoints defined using the `TRACE_EVENT` macro [38]. It is similar to Ktap, the tracing code is written as a script. The scripts are written in the SystemTap language, highly similar to the C language. These scripts are converted into C code, and then are inserted into the Linux kernel as a module. Kprobe is used to insert dynamically tracepoints into the Linux kernel. This approach, similar to Dtrace on Solaris [39], is very flexible. However, SystemTap has serious performance issues especially for collecting trace events [37]. Furthermore, its dynamic instrumentation is trap based, which adds numerous context switches.

2.4.2.5 eBPF

The newest entrant in the long list of available tracers is the Extended Berkeley Packet Filter (eBPF). BPF, which was in older Linux kernels, has been enhanced and became eBPF in the Linux 4.x series kernels. eBPF allows to do much more than just packet filtering in BPF. It enables the user to write any program and to insert it into any location in the kernel using Kprobe. It can also attach probes to available static tracepoints in the Linux kernel. eBPF scripts are compiled at runtime and executed on the small BPF VM. eBPF can now be used as a tracer, much like SystemTap or Dtrace, since it can aggregate events, analyze them, and store them in a trace. It uses the perf buffer to save events in memory and can store them as CTF events on disk. eBPF is a great tool for aggregation and live monitoring. Although an elaborate, feature-rich and easy to use tool, eBPF is still under development and does not offer the same performance, maturity and features as LTTng. Figure 2.6 depicts eBPF tc hooks, that first compile the program with LLVM and then inject it into the kernel using the bpf syscall. After the injection into the kernel, the kernel verifier checks the script for

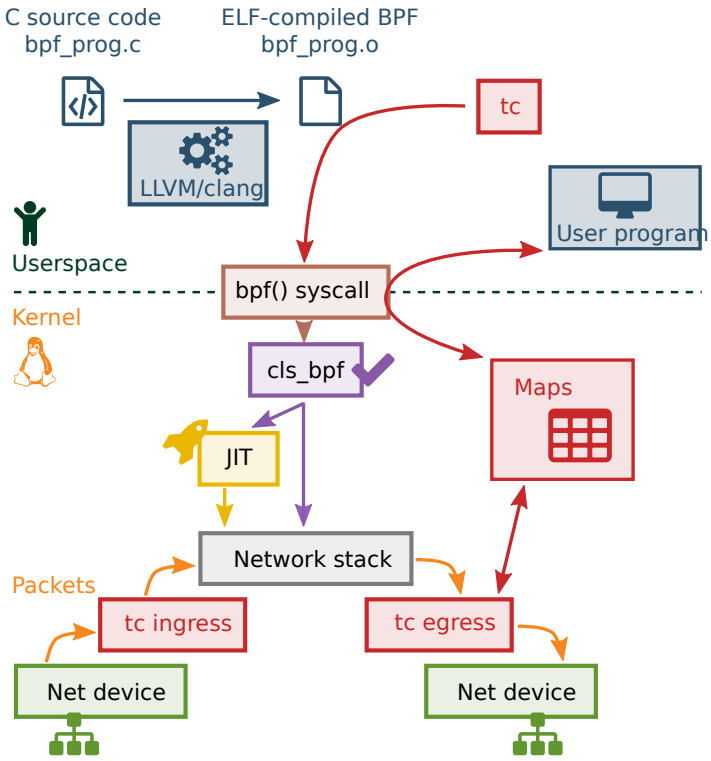


Figure 2.6 eBPF architecture for tc interface [4]

safety, attaches it to the traffic controller interface and runs it in a small VM.

2.4.2.6 Performance and Evaluation

In this project, we used LTTng since it is a system-wide and very fast tracer [37] [40]. It also provides unified kernel and user-space tracing, which is very useful in our project. LTTng can gather KVM tracepoints (Kernel tracepoints) as well as Qemu tracepoints (User-space tracepoints).

2.5 Study on existing VM analysis and monitoring approaches

Several monitoring and analysis tools have been enhanced to handle VMs. Most of them are closed-sourced and information about how they monitor VMs is a secret. Note that, to the best of our knowledge, there is no tool for debugging and analyzing different levels of virtualization without an agent (or similar internal access). In this section we examine most of the available academic and commercial approaches for monitoring VMs.

Nagios is an enterprise server and network monitoring software that mainly uses logs for monitoring systems [41]. Nagios core is an open source project and Nagios XI is a commercial version with extra capabilities. It does not have internally a mechanism to monitor the system and it relies on external plug-ins to provide its monitoring functionality. The external plug-ins are mostly basic shell scripts that interpret the data output of other daemons, or read existing logs. For example, to measure the free memory of a system it uses the `free` command and converts its output to the Nagios format. It also supports SNMP monitoring, which allows monitoring any SNMP enabled device without installing an agent. Nagios cannot provide high resolution data because of the added overhead to the target system. Any high frequency polling with this type of mechanism (e.g. command execution) could overload the target system. It collects and displays simple metrics like CPU usage, memory usage, disk usage, network usage, and the availability of a specific service or port. It is difficult to find the root cause of an issue with Nagios since it does not provide high resolution data. It ingests simple metrics, provides an alarm system, and displays a dashboard to warn the administrator of any problem detected from the metrics. It provides several views to show the data gathered from all over the network. It is extensible, since programmers can easily write external plug-ins. It scales well because the Nagios agent can automatically forward updates to servers. It does not provide any information about different virtualization layers and needs an agent inside the VM.

StackDriver is a monitoring tool that can collect plenty of metrics, events, and metadata from the Google Cloud Platform, Amazon Web Services, application instrumentation, and a variety of common application components like DynamoDB, Lambda, RDS, Amazon S3, and Apache [42]. It mostly retrieves information from application logs. Furthermore, Stackdriver Trace [43] is a distributed tracing system for the Google Cloud platform that collects data from pre-defined tracepoints inside the Google App Engine, Google HTTP(S) load balancers, and applications instrumented with the Stackdriver Trace SDKs. The traces can also be collected from Zipkin tracers via the Stackdriver Trace Zipkin Collector [44]. With the traces, it can show application latencies. It is extensible, users can add tracepoints using the Stackdriver Trace SDK. Although StackDriver mostly provides information about applications inside

virtual instances, it does not offer any mechanism to correlate this data with different events coming from different layers of the virtual environment.

SolarWinds is a network monitoring software that can detect and solve network problems [45]. It provides information about the load of cloud infrastructures and predicts the load based on the resource usage history. Then, it provides hints about the migration of VMs across the cloud infrastructure. It also aggregates some cloud metrics, like the resource usage for each process, into a monitoring dashboard for visualization. It alerts the administrator if some threshold is exceeded. It provides low resolution information and is not helpful to diagnose the root cause of problems.

Data dog is an open-source monitoring service that brings together all the data, from different applications across the network, to a single server [46]. Then, it provides a unified view to represent the data. It mainly uses other daemons or available logs to collect metrics. For example, to monitor containers, it uses Heapster to collect data. The data is stored in its database. Then, it uses the available Data Dog views to show the stored data.

Sysdig is an open-source universal system visibility tool for Linux. It can capture the system state, trace the system, save the traces, filter events and analyse traces [47]. It mainly relies on system call traces, using the available tracepoints in the Linux kernel. It provides container visibility without any plug-in or instrumentation inside. Since it uses the available Linux tracepoints, it minimizes the performance overhead and invasiveness. Recently, the Sysdig tracer had a feature added which allows putting tracepoints into any application code segment to monitor its execution time [48]. They support eBPF as an alternative to their kernel module-based architecture.

Dynatrace is an unified Application Performance Management (APM) solution [49]. It provides a full stack monitoring of applications and infrastructure, including servers, containers, and VMs. It supports Artificial Intelligence (AI) to continuously auto-detect anomalies and pro-actively pinpoint the root causes of issues. The metrics collection is based on time-based, event-based, and hybrid techniques. It can also optimize applications based on performance issues, found during the analysis. It mainly provides tracing for web application [50].

New Relic is a monitoring and analysis tool specialised in web and real-time mobile applications [51]. New Relic can collect time-slice and event data. Time-slice data are statistical metrics that aggregate over a period of time. Events data are single events with a timestamp and event body. New Relic uses event data for custom filtering or investigating the root cause of problems. It provides several dashboards and custom views to show the analyzed data. It can provide high resolution data but is more useful to find the root cause of problems in web servers.

Zenoss is an open-source monitoring and analysis tool for the cloud and physical IT environments [52]. It is highly scalable and extensible. It can monitor most modern applications like Hadoop, docker, openstack, and mySQL. It relies on existing tools to collect data using SNMP or log processing. ZenPack is an extension to Zenoss which adds the ability to build an extension for new devices or services. Zenoss provides Zenpacks for Microsoft Hyper-V, Xen, and VMware vSphere, Amazon Web services, Apache CloudStack, and Openstack. It collects data about memory, throughput, CPU and disks, and reports the data to the Zenoss Console for visualization. To monitor a virtual environment it provides libvirtSNMP that relies on libvirt-snmp as a data provider [53].

Hyperic provides more than 50,000 performance metrics for different applications and systems. vFabric Hyperic is the commercial version, and Hyperic HQ is the open-source version of Hyperic. It has an agent that must run on each physical machine and virtual instance. The Hyperic agent can automatically discover resources and applications on the platform and collect metrics. Hyperic Agent can investigate different logs and configuration files and record events. Then, the Hyperic server receives data from the Hyperic agent and stores it in the Hyperic database. In the commercial version of Hyperic, it provides trend Analysis for metrics, SLA monitoring, and exception management. It can track log events that are generated from different application like Tomcat, Apache, WebSphere, JBoss, Oracle, and mySQL [54].

The Private Cloud Monitoring System (PCMONS) is an open-source solutions for managing and monitoring private clouds. It is an extensible and modular monitoring system and is compatible with Eucalyptus, Nagios, Zabbix,and Opennebula. It is implemented as several modules as follows [55]:

Node information gatherer: Gathers metrics from different VMs and sends them to the Cluster Data Integrator.

Cluster Data Integrator: Gathers, organizes, and prepares the collected data for the monitoring data integrator. It removes the unnecessary data before the transfer to the monitoring data integrator.

Monitoring Data Integrator: Stores the data to a database to be used by the Configuration Generator.

VM Monitor: Sends and executes scripts inside VMs, for example a script to measure the free disk space on a VM.

Configuration Generator: Converts the stored data to a configuration file for the visualization tool.

Monitoring Tool Server: In the case of multiple resources to monitor, it gathers all of the data in one place.

User Interface: Implements an interface to different visualization tools.

Database: The database stores the Configuration Generator data and the Monitoring Data Integrator data.

PCMONS provides a flexible and extensible architecture to monitor different applications in a virtual environment. In order to extend PCMONS, the user can write external plug-ins.

OpenNebula is an open-source scalable platform for managing distributed datacenter infrastructures [56]. It also provides the OpenNebula monitoring subsystem that gathers basic metrics from virtual instances and Hosts. It periodically sends the data gathered via UDP to the monitoring frond-end, using a daemon named collectd. It is extensible since users can write custom probes and send new metrics to the frond-end [57].

Ganglia is an open-source scalable metrics collection system for large infrastructures. It is very lightweight (add less than 1% CPU overhead [58]) and keeps all the data in memory [59]. It consists of [60]:

gmond: The ganglia monitoring daemon (gmond), a lightweight service, that uses a simple listen/announce protocol via the eXternal Data Representation (XDR) to collect the different states of systems and their processes. The collected metrics are simple metrics available in /proc/.

gmetad: The ganglia meta daemon (gmetad) is a data collector that receives data from gmond and stores it in a database.

gmetrics: The ganglia metric (gmetrics) is a command-line tool for which the user can write custom metrics. For example, the user can send the CPU temperature to ganglia using the command below:

```
$> gmetric --name temperature --value ‘cputemp‘ \
--type int16 --units Celcius
```

gstat: The ganglia stat tool queries the monitored data from gmond and shows it in the terminal [61].

In [58], experiments were conducted to validate the scalability of Ganglia. Ganglia is extensible since programmers can easily add metrics into their application using an embedded gmetric. It accepts condition for its filter to report metric changes. Ganglia has some limitations. It cannot provide fine-grained information about systems and it is based on sampling existing metrics in systems. The sampling rate is adjustable, but high frequency sampling could overload a system. The provided metrics are fairly general and do not include specific metrics related to VMs like the virtualization overhead, or hypervisor memory usage.

In [62], [63], the authors implemented guest-wide and host-wide profiling, using Linux perf to sample the Linux kernel running in KVM. It allows profiling applications inside the guest by sampling the program counter (PC) with perf. After PCs are retrieved by perf, they are mapped to the binary translated code of the guest to find out the running time for each function inside the VMs. To have a more precise profiler, the sampling rate should be increased, which incurs more overhead on the VMs.

Khandual *et al.* in [64] presents a Linux perf based virtualization performance monitoring tool for KVM. They benefit from counting the occurrences of different events in the guest to detect anomalies. In their work, they need to access each VM, which is often not possible because of security issues and overhead. While increments to values of some event counters may be an indicator of problems in the guest, it cannot show the exact problem and the associated time.

In [65], the authors developed a CPU usage monitoring tool for KVM relying on *"perf kvm record"*. By profiling all CPUs, they could monitor the CPU usage of VMs and the total CPU usage of the hypervisor. In their work, they needed to profile the guest and host kernel at the same time. They were capable of finding the overhead introduced by virtualization for VMs as a whole, but they cannot measure it for each VM separately.

VMon [66] uses hardware PMU to detect VM interference. In this paper, the authors investigate the relationship between the LLC miss rates and VM interference. VMon collects hardware PMU data and feeds it to a predictor to quantify the performance degradation. In the results section of the paper, the authors show that the performance interference of CPU-intensive VMs is highly correlated with LLC miss rate. Another work [67] investigates the impact of resource sharing on VMs. The authors have shown that LLC and the virtual CPU (vCPU) to physical CPU (pCPU) ratio are the main reasons for performance degradation. Other metrics that could reduce the performance are the "Size of File" and "Block Size". Co-locating several VMs in a host can reduce the LLC hit rate because of the numerous context switches between vCPUs. Furthermore, if the sum of Block size and Size of File is larger than the LLC, there could be a major performance degradation. The LLC miss rate is

an acceptable indicator of performance reduction for CPU-intensive VMs since the only stall time is for fetching data from main memory. In contrast, for I/O intensive VMs, the CPU is not utilized all the time and the performance reduction could be caused by I/O delay.

PerfCompass [68] is a VM fault detection tool for internal and external faults. It can detect if the fault has a global or local impact. As part of their implementation, they trace each and every VM with LTTng [34]. The data is eventually used to troubleshoot VMs and find out problems like I/O latency, memory quota problems and CPU quota problems. Their approach, however, needs to trace each VM, which significantly increases the overhead for VMs. Their approach can be ported to nested VMs by tracing each nested VM. Nonetheless, the overhead of tracing and analysing each VM is significant.

PerfGuard [69] is a production-run performance diagnosis tool. It analyzes the binary of an application and generates a performance profile of the application. It partitions the application code into program execution partitions (also called units). Then, it clusters similar units based on their control flow and embeds several guards into the application binaries. PerfGuard examines each execution unit and compares it to some threshold to find problems. It can only find a limited number of problem types. They have not provided any information about VM centric applications.

A formal study of the prediction-based proactive load balancing for VM migrations was done by Anju, Bala et al. [70]. In their approach, resource utilization parameters like CPU, RAM, and Disk I/O are continuously sampled and stored in a file. Moreover, they test different machine learning approaches on this data to compare and minimize the error. Then, they compare the resource utilization with a threshold. If the load is above the threshold, the host is classified as over-utilized. Then, a VM will be selected based on the inter-correlation coefficient approach. The algorithm identifies the VM which has the highest correlation with resource utilization, and selects it for migration to another host. Conversely, if the load is less than the threshold, then the host is classified as under-loaded. In this case, all VMs will be migrated to another host in order to switch off the under-loaded host. The metrics used in this project could be more extensive, improving the experiment. They could group VMs based on their load and then migrate the VM which is using more resources among the ones over-utilized.

vSpec [71] [72] is a workload-adaptive OS customization tool that can modify VM kernels for better performance, based on their workload. They have separated VMs into five different categories according to their resources consumption. These five classes are: CPU-intensive, Memory-intensive, I/O-intensive, network-intensive and compound. 65 different metrics are being collected from /proc/, once per second, for system, CPU, memory, disk, network,

cache, etc. They have implemented a workload classifier using a modified version of the Support Vector Machine (SVM) algorithm, named Training Set Refreshed SVM (TSRSVM). Then, they customize the operating system to assign extra capacity adaptively and to reduce operations unnecessary for the workload.

In [73], the authors proposed a technique to investigate different VMs states. They could find the preempted VMs along with the preemption cause. In their case, each VM and the host kernel were traced. After tracing, they synchronize the trace from each VM with that from the host. Then, they search through all threads to find the preempted threads.

The only analysis and monitoring tool for nested VMs was presented in [74] [75]. They provide a fused virtualized system analysis that integrates traces from different virtualization layers. It can show the control flow of processes inside nested VMs in different virtualization levels. Synchronization is an essential part of their tool, since traces are gathered from different layers with different internal clocks. They can find the preempted nested VMs along with the preemption cause. In their case, they trace each VM, nested VM and also the host kernel. After tracing, they synchronize the trace from each VM with that from the host. Then, they search through all threads to find preempted threads. In an interesting use-case, they compared the time needed to wake-up a process inside a normal VM and a nested VM. The wake-up time for the nested VM is almost 5 times the wake-up time for the normal VM. Indeed, nested VMs incur a much larger virtualization overhead because it switches between the different virtualization layers repeatedly. In [76], the authors proposed a method to reduce the number of transitions between the different virtualization layers using VM exit prediction. In their method, the VMM inspects privileged instructions in guest code. Then, they use binary translation to generate code dynamically and inject them into the guest code. They have shown that this method could save up to 50% of the transition costs.

In [77], a monitoring architecture for tracking VM software is proposed. Their architecture needs to install an agent inside the VM to obtain VM application information. The detailed information about the VM (like resource usage, VM name, etc.) is extracted in a transparent manner from the hypervisor level.

Trihinas et al. in [78] propose a platform-independent and distributed monitoring architecture. Linux tools like `iostat, vmstat, and top` are used to collect statistics about the application running inside the VM [79] [80]. Then, the daemon inside the VM parses the output of the existing Linux tool and sends the gathered data back to the central agent, to be stored in a database.

An agent-less monitoring architecture is proposed by Calero et al. in [81]. It uses existing approaches from OpenStack to provide basic metrics for each VM. Their method is dependent

on OpenStack and cannot be used on other platforms for cloud computing. Another agentless technique to analyze the VM behaviour is proposed in [82]. They used an Call Setup Delay as an end-to-end to scale up and down the number of CPUs. This method does not provide any reliable information about other available resources.

Apart from obtaining metrics and information from the host hypervisor level, Virtual Machine Introspection (VMI) is another technique to analyze the running state of VMs without installing an agent inside the VM [83]. In the VMI technique, the whole VM memory will be dumped and then analyzed. This technique is used for security threats analysis but could provide information about running processes. Several VMI techniques are proposed in [84] [85] for malware and security threat detection. Analysing the memory of each VM takes a significant time. It could be more challenging to analyse the memory of thousands of VMs. Furthermore, none of the VMI tools target VMs performance analysis in terms of resource usage and contention.

The term Critical path analysis is being widely used for industrial projects to identify the sections to optimize. Similarly, in software engineering, critical path analysis is a way to detect and analyse the resource bottlenecks. The critical path identified may be used by the administrator of the infrastructure provider to tune the performance of VMs, in order to provide a better quality of service.

vPath [86] is a method that can identify the request dependencies of VMs. The method is specific to Linux systems and some modification are required at the hypervisor level. The VMM must be modified to intercept each system call, in order to obtain the VM dependencies. Intercepting each system call adds further overhead, since it needs to switch between the guest and host modes back and forth. It also does not provide any information about the resource bottlenecks and why the processes are waiting inside the VM.

The wait analysis of distributed systems is proposed by Giraldeau et al. in [87]. They presented and implemented an algorithm to recover the active path for the processes using kernel traces. They can distinguish different states like wait for disk, network, timer, and tasks. Although their method could be applied to processes inside virtual machines, it needs to access the VMs and trace each VM individually. Then, they have to synchronize the traces and search through all the threads to discover the active path.

Canali *et al.* [88] exploited the correlation between the usage of multiple resources to determine which VMs were following the same behavioral pattern. Then, the K-Means approach was applied to cluster together similar VMs in an IaaS cloud data center. Their method sampled resource utilization metrics, such as CPU, memory, disk, and network usage, with no assumption on the knowledge of the processes running in the VMs.

In [89] and [90], a Smoothing Histogram-based clustering (SH-based clustering) approach was proposed, in order to group VMs showing similar behavior in a cloud environment. The monitoring metrics were periodically collected from the VM resources usage using the hypervisor APIs. They applied the Bhattacharyya distance to measure the similarity between two VMs based on the probability distributions of resource usage. A spectral clustering based approach was employed to categorize VMs into distinct groups. Their work focused on monitoring scalibility by selecting few representative VMs as the VMs closest to the cluster centroids. Furthermore, in [91], the authors employed a similar strategy to propose a VM placement technique, namely class-based placement. The class-based method leveraged the knowledge of VMs classes to select few representatives and reduce the size of the problem to be solved. The solution obtained was then replicated as a building block to solve the global VM placement problem.

In another work [92], a K-Means clustering based approach was proposed to monitor the cloud security against various anomalies, like intensive resource usage and malware attacks. System-level metrics, such as CPU, Memory, Disk, and network usage were collected for every monitored VM. In [93], the authors included the fine-grained information of each process in a VM for malware detection in cloud infrastructures. The proposed method trained a 2D Convolutional Neural Networks (CNNs) model on performance metrics gathered from processes running on the VMs.

Zhang *et al.* in [94] proposed a density-based clustering method (DBSCAN) for abnormal behavior detection in large scale clouds. The performance related metrics such as CPU usage, memory utilization, and disk usage were collected by BOSH agents. An entropy-based feature selection technique was used to reduce the data dimensionality.

In [95], the authors collected kernel event traces of all active processes in a cloud infrastructure and then pushed them into a central log store. An event sketch modeling module converted the raw event traces into a group of kernel events having causality relationships. A set of statistical and data mining analysis tools were offered to summarize the event sketches and perform cloud performance diagnosis.

Tracing VMs was used in [96] to detect Denial of Service (DoS) attacks in cloud infrastructures. A trace abstractor created high-level features from the statistics of low-level events, such as CPU usage and number of HTTP connections. The Support Vector Machine (SVM) algorithm was then applied to detect changes in VM behaviors.

2.6 Summary of Literature Review

In the past decade, various VM analysis tools have been developed. In this literature review, we explained the state-of-the-art techniques being used in this domain. We can conclude that the field of VM analysis using tracing is a relatively new and highly relevant area. It provides numerous research opportunities for enhancing the analysis techniques and reducing the overhead.

The two main developments which motivated the current research, are presented in [87] and [73]. In [87], Giraldeau et al. present a technique to find the active path for the threads through different distributed machines. Gebai *et al.* in [73] proposed a technique to investigate different VM states. The authors could find the preempted VMs along with the preemption cause. Both need to have an agent inside each VM for tracing. In contrast, the work proposed in this thesis is based on an agent-less technique, but still provides the same information.

We also realized that there is an obvious lack of efficient analysis techniques for nested VMs. To the best of our knowledge, there is no pre-existing efficient technique to analyze the performance of any level of VMs. Our technique can uncover many issues inside VMs, without internal access. Moreover, compared with other existing solutions, our proposed method incurs less overhead, and is easier to deploy, since it limits its data collection to the host hypervisor level.

CHAPTER 3 RESEARCH METHODOLOGY

As discussed briefly in Chapter 1 and as illustrated in figure 3.1, our work focused on VM analysis. The first area was analysing how Intel x86 (Intel-VT) CPU cores support the virtual Machine eXtensions (VMX) instruction set, in order to find out how they differentiate between processes inside the VM, marked as [P1]. Then, we proposed a new technique to identify the VM processes without an in VM agent. While studying industrial workloads, we realized that a VM executing another VM (Nested VM) was a very common use case in industry and often exhibited performance problems. In order to propose a general architecture for analyzing VMs, we then proposed a new technique to detect nested VMs [P2]. Thereafter, our work focused on analyzing the execution flow of VM processes along with detecting the different states for virtual CPUs [J1, J2]. Then, based on our vCPU state analysis and execution flow analysis technique, we proposed new efficient algorithms to cluster VMs based on their workload [J3]. This is described briefly in the following sub-sections.

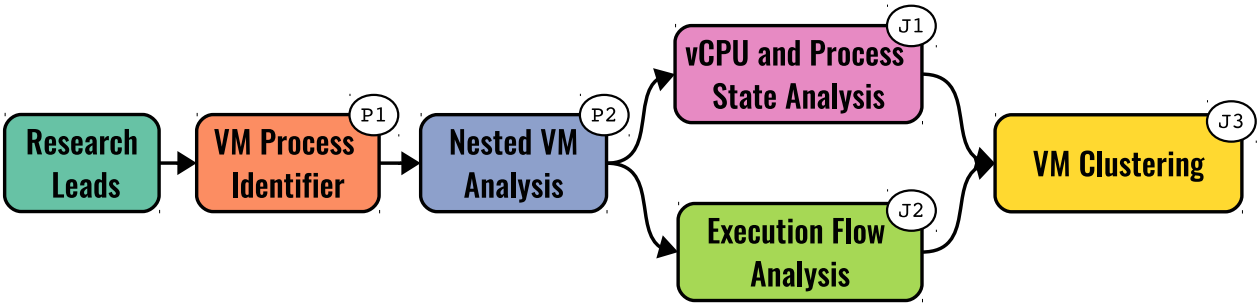


Figure 3.1 Research milestones and progression

3.1 VM Process Identifier

The process identifier (PID) and process name of each thread inside the guest is not directly accessible from the host. Therefore, we studied how pysical CPUs differentiate between the processes inside VMs. Modern Intel (and similarly AMD) CPUs support two execution modes and a special instruction set named virtual Machine eXtensions (VMX). The VM code executes in `non-root` mode and the hypervisor code runs in `root` mode. The transition between `root` mode and `non-root` mode is called a VMX transition. Furthermore, non-privileged VM instructions are executed as `non-root` mode, and privileged instructions are executed as `root` mode (at a higher privilege level). In each transition, the environment

specification of the VMs and hypervisor are stored in an in-memory structured named Virtual Machine Control Structure (VMCS). The VMCS structure contains several fields including: **exit_reason:** the reason for transition from `non-root` to `root`, **injected_interrupts:** the injected interrupts to the VM, **CR3:** pointing to the page directory of a process in VM, and **SP:** pointing to the stack of the thread inside the VM. We found that CR3 and SP can identify the process and thread, respectively. In order to retrieve these two identifiers, we created a new tracepoint, `vcpu_enter_guest`, for the host using kprobe.

3.2 Nested VM Analysis

Nested VMs are also supported by Intel and AMD. In a nested VM, there are two hypervisors. Executing any privileged instruction by any level nested VM returns to host hypervisor level (L0). In this case, the VM hypervisor (L1) has the illusion of running the code of nested VM (L2) directly on a physical CPU. However, privileged instructions of nested VMs must be handled by the highest privileged level. During our literature review we found that there was no technique to analyze such a complex architecture.

In order to analyze nested VMs, we proposed the new Nested VM State Detection (NSD) algorithm that can find the CR3 of the guest hypervisor (L1). As mentioned before, the guest hypervisor has the illusion of executing the VM code directly on a pCPU. As a result, in order to resume or launch a VM, it uses the special VMX instructions `resume` or `launch`, respectively. Executing these instructions causes an exit to the host hypervisor from L1, and the algorithm marks the last CR3 as the CR3 of the hypervisor in L1. Furthermore, the next CR3 is the CR3 of the process inside the nested VM, since the nested VM is executed directly by host the hypervisor. This part of our work, corresponding to milestone [P2], has been presented in Article 1 (Chapter 4) where we propose the Nested VM state detection algorithm. Then, we also introduced a new approach to detect over-commitment in any level of virtualization.

3.3 vCPU and Process State Analysis

As mentioned before, the running process state alternates between the VMX `root` and VMX `non-root` states. On the other hand, processes waiting for a resource are in the Preempted and wait_for_x states. Figure 3.2 depicts different states for vCPUs and Processes. A process could be executing in any level and could be preempted in any virtualization level. As a result, a process in any level could wait for a resource. We found that detecting the different states of vCPUs and processes in any level could help the IaaS administrator to

investigate the root cause of latencies in VMs. We proposed an algorithm to uncover the vCPU and process states for arbitrary nesting depths [J1]. The algorithm uses the injected interrupt to reveal the reason for waiting, as well as `vm_exit` event to find the transition between VMX `root` and VMX `non-root` states.

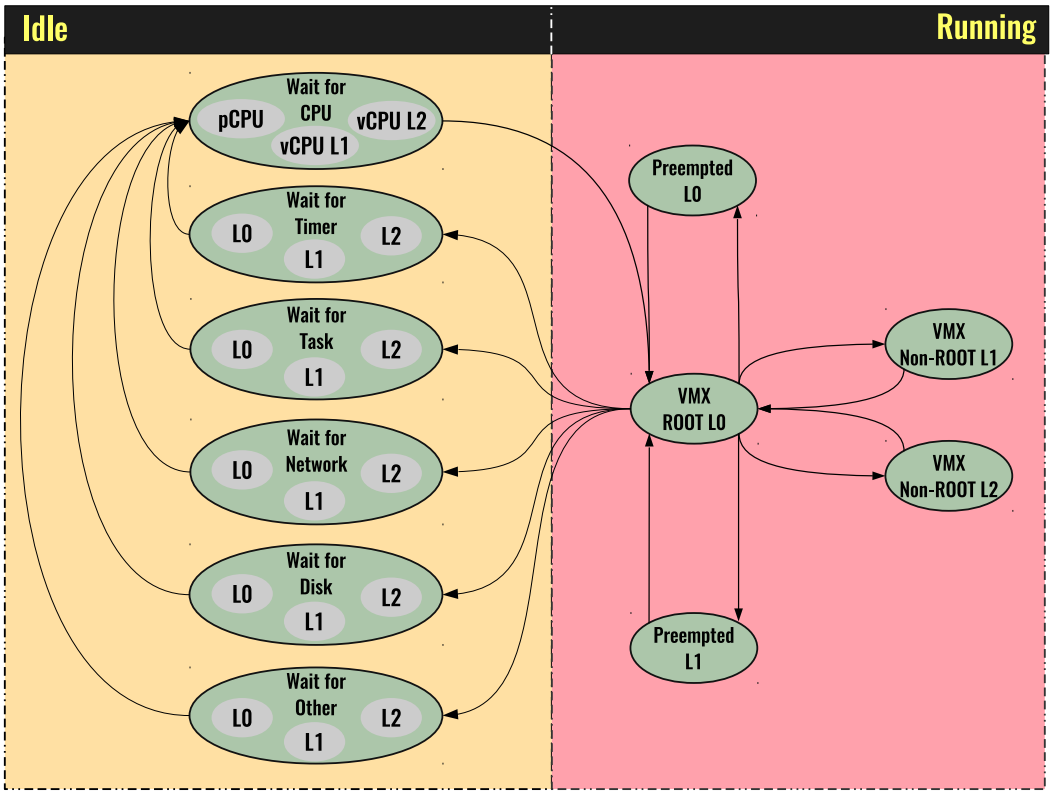


Figure 3.2 Virtual Machine Process State Transition

3.4 Execution Flow Analysis

Tracking and finding dependencies between VM processes shows why and when a process waits. A process could wait for different reasons. A process could wait for a timer to fire. If a process is woken up by another process, it indicates that it was waiting for another process to finish its task. Also, a process could wait for a device. As an example, if a process is woken up by network interrupt, it indicates that it was waiting for an incoming packet. The proposed solution aims at supporting the IaaS provider in understanding the resources consumption, and the dependencies among processes, even across VMs. We proposed and implemented the Host-based Execution-graph Construction algorithm to determine the dependencies among the processes and different resources [J2]. The algorithm builds a two dimensional process dependency graph using the injected interrupts and `vm_entry` and `vm_exit` events.

3.5 VM Clustering

The efficient operation and resource management of multi-tenant data centers hosting thousands of services is a demanding task that requires precise and detailed information regarding the behaviour of each and every virtual machine (VM). Often, coarse measures such as CPU, memory, disk and network usage by VMs are considered for grouping them onto the same physical server. Indeed, detailed measures would require access to the guest operating system (OS), which is not feasible in a multi-tenant setting.

We proposed the host level hypervisor tracing as a non-intrusive mechanism to extract useful features that can provide a fine grain characterization of VM behaviour. In particular, we extract VM blocking periods as well as virtual interrupt injection rates to detect multiple levels of resource intensiveness. In addition, we consider the resource contention rate, due to other VMs and the host, along with reasons for exit from non-root to root privileged mode, to reveal useful information about the nature of the underlying VM workload. We also use tracing to get information about the rate of process and thread preemption in each VM, extracting process and thread contention as another feature set. We then employ various feature selection strategies and assess the quality of the resulting workload clustering.

3.6 Experimentation

For each of the research milestones, extensive experimentation was performed to assess the performance and overhead of analysing VMs. Then, we compared our approaches with existing methods. We reproduced the test results multiple times to insure that our observations were statistically significant.

Figure 3.3 shows the architecture of our implementation. We used a very lightweight tracing tool called Linux Trace Toolkit Next Generation (LTTng) [1]. LTTng is an open source tracing toolkit for Linux that allows to to gather events related to interactions between user space and kernel space. Additionally, the Linux kernel and KVM module are instrumented by specialized static tracepoints, that LTTng collects and sends to the trace analyser. The tracepoints used for workload analysis are shown in Table 3.1. `sched_wakeup` and `sched_switch` are existing Linux kernel tracepoints and are related to the process scheduler. `vm_exit` and `vm_inj_virq` are existing KVM module tracepoints and are related to the virtualization technology. In order to complete our analysis, we create a new tracepoint, `vcpu_enter_guess`, for the host using kprobe, which can retrieve CR3 and SP from VMCS at each VM entry. Note that all our tracepoints are at the host level and we do not need to access the guest level. Consequently, we impose a lower overhead on the VM.

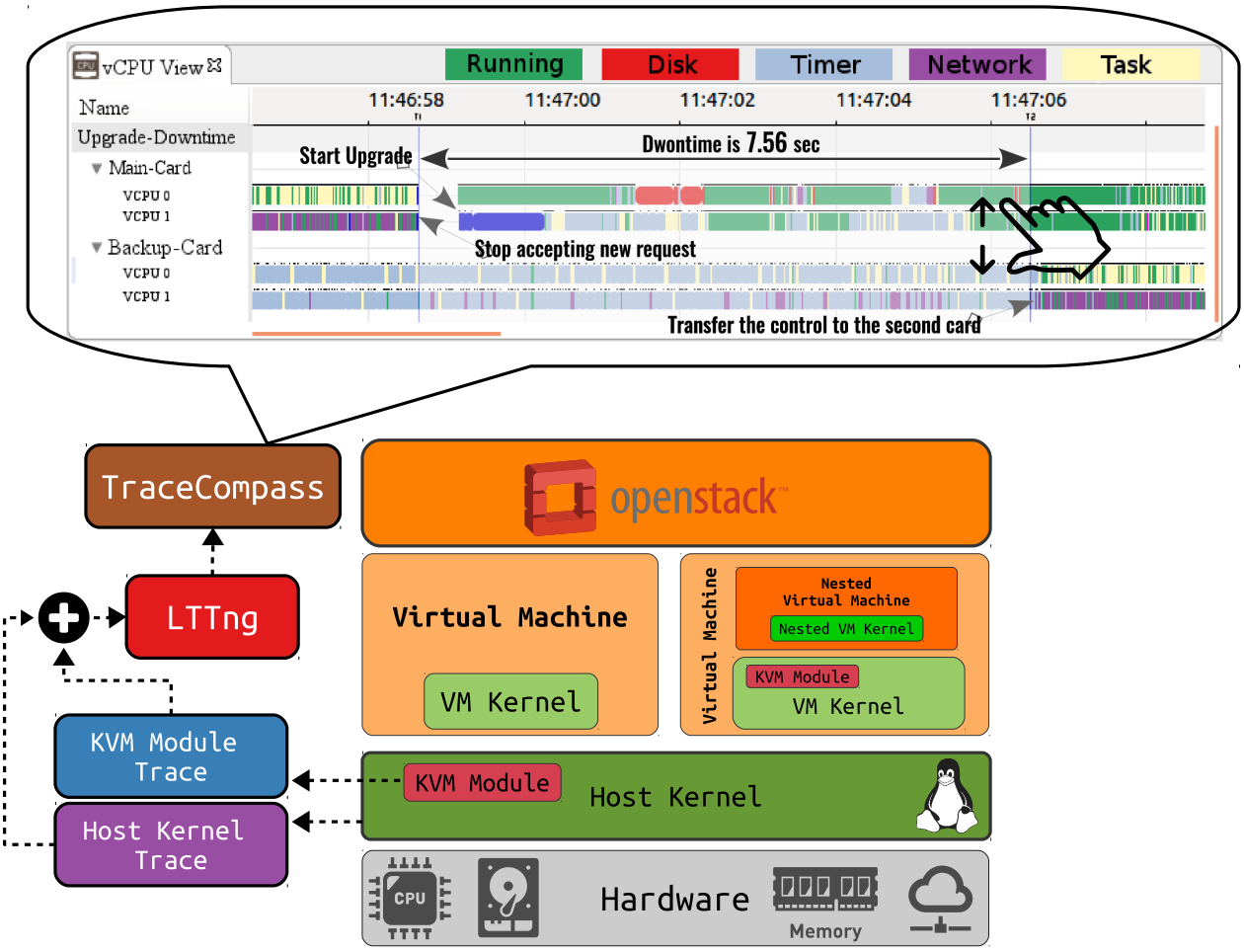


Figure 3.3 Architecture of our implementation

We implemented our algorithms in Trace Compass [2] as a separate module. Trace Compass is an open source tool for viewing and analyzing traces. Although, Trace Compass has several pre-built analysis modules to extract useful information from huge traces, it does not provide agent-less VM analysis.

We extract some useful information from our collected traces and store it in a database named State History Tree (SHT). The SHT is a tree shaped disk database used to store the state of various components (represented as integer, long or string values) associated with a time interval.

The new developments outlined in this chapter are detailed in the subsequent chapters, each being a research article.

Table 3.1 Needed Tracepoints for our vCPU state detection and Execution path Analysis

Tracepoint	**Description**
`sched_wakeup`	Wakeup and resume a task
`vcpu_enter_guest`	vCPU enters guest mode
`vm_exit`	vCPU exits guest mode
`vm_inj_virq`	Inject virtual interrupt into VM
`sched_switch`	Scheduling out or in a process

CHAPTER 4 ARTICLE 1: "VIRTFLOW: GUEST INDEPENDENT EXECUTION FLOW ANALYSIS ACROSS VIRTUALIZED ENVIRONMENTS"

Authors

Hani Nemati, Member, IEEE and Michel Dagenais, Senior Member, IEEE

Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

Published in IEEE Transactions on Cloud Computing

Reference as H. Nemati and M. R. Dagenais, "virtFlow: Guest Independent Execution Flow Analysis Across Virtualized Environments," in IEEE Transactions on Cloud Computing. doi: 10.1109/TCC.2018.2828846

4.1 Abstract

An agent-less technique to understand virtual machines (VMs) behavior and their changes during the VM life-cycle is essential for many performance analysis and debugging tasks in the cloud environment. Because of privacy and security issues, ease of deployment and execution overhead, the method preferably limits its data collection to the physical host level, without internal access to the VMs. We propose a host-based, precise method to recover execution flow of virtualized environments, regardless of the level of virtualization. Given a VM, the Any-Level VM Detection Algorithm (ADA) and Nested VM State Detection (NSD) Algorithm compute its execution path along with the state of virtual CPUs (vCPUs) from the host kernel trace. The state of vCPUs is displayed in an interactive trace viewer (TraceCompass) for further inspection. Then, a new approach for profiling threads and processes inside the VMs is proposed. Our proposed VM trace analysis algorithms have been open-sourced for further enhancements and to the benefit of other developers. Our new techniques are being evaluated with workloads generated by different benchmarking tools. These approaches are based on host hypervisor tracing, which brings a lower overhead (around 1%) as compared to other approaches.

4.2 Introduction

Virtualization is an emerging technology that enables on-demand access to a pool of resources through a Pay as Use (PaU) model. Sharing resources plays an important role in cloud computing. Many enterprises are beginning to adopt VMs in order to optimally utilize their resources. Despite its merits, debugging, troubleshooting, and performance analysis of such large-scale distributed systems still are a big challenge [97].

This challenge often becomes more complicated when, because of security issues, the infrastructure provider does not have access to the VMs internally. Thus, in the case of performance issues, the Infrastructure as a Service (IaaS) provider cannot provide useful insight. Moreover, the IaaS provider scheme could give the cloud user the ability of managing and using their own hypervisor as a VM (Nested VM). In this case, the diagnosis of added latency and response time of Nested VMs is quite complex, due to different levels of code execution and emulation.

Contemporary use of nested VMs is more for the purpose of software scaling, compatibility, testing and security. In addition, many network services may be virtualized (as the main goal of NFV) and hosted on nested VMs. Software as a Service (SaaS) providers are the best clients of nested virtualization. SaaS providers encapsulate their software in a nested VM on an existing cloud infrastructure (e.g., Google Cloud and Amazon AWS). To show how recently nested VMs are becoming important, we analyzed the latest commits for KVM in the Linux Kernel (from Kernel version 4.9 to 4.10). As the latest commits [98] show, 51% are directly related to adding new features or improving performance for nested VMs.

On the other hand, cloud applications become more complex, where their workload may vary due to time and geographic location. Thus, cloud computing lets the end-users scale resources quickly. For example, Figure 4.1 shows the execution latency for the same workloads. As shown, the execution time is not the same for each run (e.g., 342 ms, 699 ms, 351 ms). The average for 100 executions of the same workload is 443 ms, with a standard deviation of 116 ms. As a real use case, we did the same experiment on an Amazon EC2 t2.micro instance. In this experiment, the maximum and minimum execution time were 786 ms and 448 ms, respectively. We expect the execution time for the same workload should be almost the same. The execution time for the same task is sometimes different and the VM has the illusion of running all the time. To investigate the cause of latency, we traced the VM, but we could not find any meaningful information. In this case, the cause might be a physical resource contention, on the host, which is not visible by tracing at the VM level.

However, the analysis of the variation in response time of VMs is quite complex due to

the cost of monitoring, security issues, and different levels of code execution. Hence, for such a complex environments, there is a need to elaborate more sophisticated techniques for performance analysis of virtualized environments.

This paper proposes an efficient technique to analyze the performance of any level of VMs, in any level of virtualization, without internal access. We propose, implement, and evaluate a technique to detect performance reductions along with many useful metrics for analyzing the behavior of VMs. In particular, we trace the host hypervisor to detect VMs and nested VMs and the different states of their running processes and threads. Our technique can investigate the root cause of latency in the VM by just tracing the host. A massive amount of information is buried under the vCPUs of the VMs. This information could be revealed by analyzing the interaction between the host hypervisor, VM hypervisors, and nested VMs. Our technique leverages existing static tracepoints inside the host hypervisor along with our new added tracepoint, to convert the tracing information to meaningful visualization.

To the best of our knowledge, there is no pre-existing efficient technique to analyze the performance of any level of VMs in any level of virtualization. Notably, the required technique should troubleshoot unexpected behavior of VMs in any level, without internal access due to security issues and extra overhead.

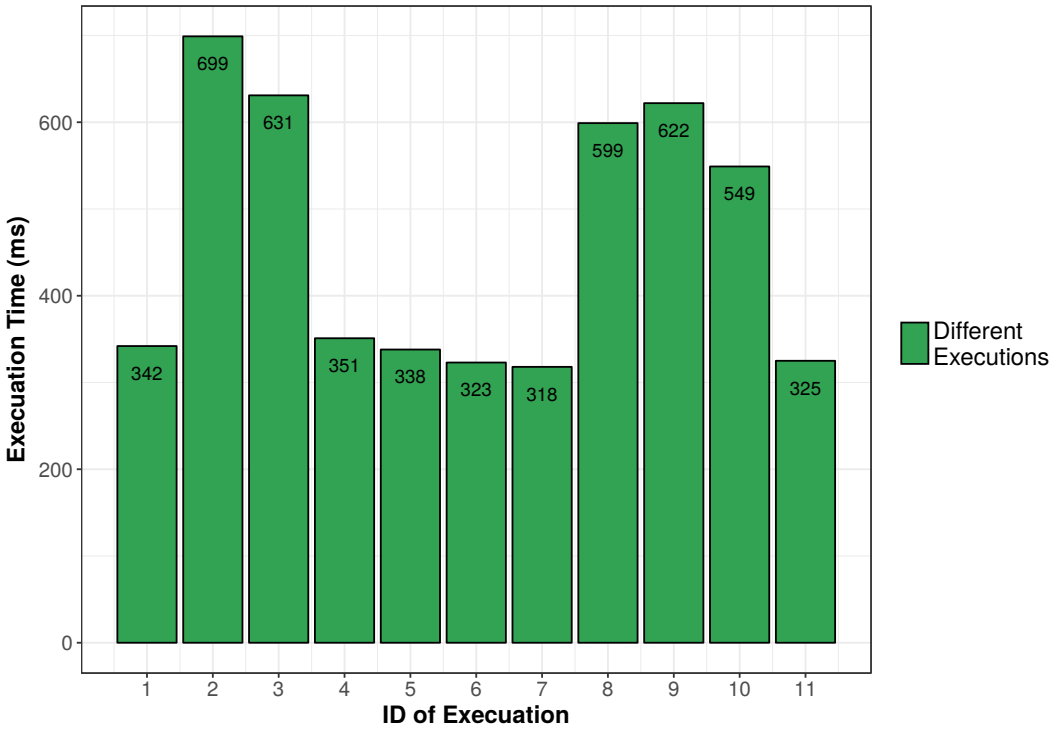


Figure 4.1 Execution latency (ms) for the same workload

Our main contributions in this paper are: **First**, our methods limit their data collection to the physical host level, without internal access to VMs. All the tracing and analysis parts are hidden to the VMs and also nested VMs. As a result, VMs and nested VMs are not being accessed during the analysis. This is critical since, in most situations, due to security reasons, accesses to the VMs are restricted. **Second**, our analysis, which is based on host hypervisor tracing, enables the cloud administrator to differentiate different states (e.g., Executing Nested VM code, Guest Hypervisor Code, and Host Hypervisor Code) of nested VMs. **Third**, we propose a method to detect different states of processes and threads, not only inside the VMs but also inside nested VMs. This method can profile processes and threads inside the VMs and nested VMs. **Fourth**, we evaluate the cost of added overhead (around 1%) due to tracing the host and compare it to other existing approaches.

Fifth, we experiment on actual software (e.g., Hadoop) to study the behavior of VMs regarding the overcommitment of resources and other possible problems. Furthermore, we implemented different graphical views as follows: first, a graphical view for vCPU threads from the host point of view. It presents a timeline for each vCPU with different states of the VM; second, we developed a graphical view for nested VMs which shows the vCPU threads of nested VMs with its level of code execution and states. Third, we implemented a process view which shows the different states of threads, along with the execution time for each.

The rest of this paper is organized as follows: Section 4.3 presents a summary of other existing approaches for analyzing and debugging VMs. Section 4.4 introduces some background information about nested virtualization technology and presents the different states of applications inside the nested VMs and their requirements. Section 4.5 presents the algorithm used to detect nested VMs from vCPU threads of the VM. It also explains how we can find the different states of vCPUs of VMs and nested VMs. A new approach for thread analysis inside the VMs, without tracing the guest OS, is illustrated in this section. Section 4.6 presents our experimental results along with the architecture used in our paper. We also propose other methods for the performance analysis of VMs and Nested VMs, and we compare these approaches in terms of overhead, ease of use, and limitation, in section 4.7. Section 4.8 concludes the paper with directions for future investigations.

4.3 Related Work

In this section, we survey the available tools for monitoring VMs and briefly propose an approach for using them to analyze nested VMs.

Until recently, virtualization on commodity servers used to be complex and slow due to ma-

chine emulation and on-the-fly binary translation of privileged instructions. In due time, with the introduction of Hardware-assisted virtualization (Intel-VT and AMD-V), the overhead and complexity were reduced. It allows the execution of non-privileged VMs directly on the physical CPU. It also provides better memory and I/O management for assigning I/O devices to VMs.

Nested VMs are also supported by Intel and AMD processors. There are two types of hypervisors that support Nested VMs: those that are Closed source and those which are Open source. Information about how Closed source hypervisors (e.g., hyper-V and VMware's hypervisor) work is not public. VMware is one of the leading hypervisors. Workstation 8, Fusion 4, and ESXi 5.0 (or later) offer nested virtualization. The client can run guest hypervisors at level 1 [99]. As a result, VMware supports one level of nested virtualization. Hyper-V, like VMware, supports one level of virtualization. Some hypervisors are Open source like Xen, and KVM. Nested VMs were suppoprted on Xen since the introduction of HVM guest in Xen 3.0. Similar to KVM, Xen supports one level of nested VM [100]. Kernel-based Virtual Machine is one the most used hypervisor and is well supported by Linux. It also supports one level of nested virtualization.

CloudVisor [101] provides a transparent security monitor for the whole VM by using a nested VM. It adds an extra layer to the Virtual Machine Monitor (VMM) to intercept privileged instructions and to protect the VM with cryptography. McAfee Deep Defender [22] is another example of nested VM use. For security reasons, it has its own Virtual Machine Monitor. Furthermore, one of the features in Windows 7 for professional and ultimate editions is the XP mode [23]. In this mode, a VM runs Windows XP for compatibility reasons. Thus, Windows 7 users can execute Windows XP applications without any change. Correspondingly, the XP mode will be run in a nested VM if Windows 7 is running in a VM.

Ravello systems [20] has implemented a high-performance nested virtualization called as HVX. It allows the user to run unmodified nested VMs on Google cloud and Amazon AWS, without any change whatsoever. Nested virtualization is also being used for Continuous Integration (CI). CI integrates code, builds modules and runs tests when they are added to the larger code base. Because of security and compatibility issues, the building and testing phases should be run in an isolated environment. CI service providers can execute each change immediately in a nested VM.

Several monitoring and analysis tools, like AWS CloudWatch [102] and Cisco Cloud Consumption Service [103], have been enhanced for practical use. Most of them are closed-source and information about how they monitor VMs is a secret. Based on our knowledge, there is no tool for debugging and analyzing different levels of virtualization without internal access.

AWS CloudWatch [102] is a closed-source performance monitoring tool that can report CPU, Network, Memory, and Disk usage for Amazon EC2 cloud. Ceilometer [104] is the metering, monitoring and alarming tool for OpenStack. Both tools provide basic metrics for physical CPUs like CPU time usage, average CPU utilization, and number of vCPUs for each VM. Although they could provide information for one level of virtualization, in case of nested VMs they can not provide any information.

Novakovic *et al.* [105] relies on some performance counters and Linux tools like *iostat* for monitoring VMs. Linux provides some performance monitoring tools, such as *vmstat* and *iostat*, which gather statistics by reading *proc* files. Parsing the output data from these Linux tools adds overhead. In the case where these tools for nested VMs are used, the added overhead could be significant.

In [62] [63], the authors implemented guest-wide and host-wide profiling, which uses Linux *perf* to sample the Linux kernel running the KVM module. It allows profiling applications inside the guest by sampling the program counter (PC) from perf. After PCs are retrieved from perf, they are mapped to the binary translated code of the guest to find out the running time for each function inside the VMs. To have a more precise profiler, the sampling rate should be increased, which causes more overhead to the VMs. Khandual *et al.* in [64] presents Linux perf based virtualization performance monitoring for KVM. They benefit from counting the occurrence of different events in the guest to detect anomalies. In their work, they need to access each VM, which is not possible most of the time because of security issues and overhead. While increments to values of some event counters could be an indicator of a problem in the guest, it cannot show the exact problem and associated time. In [65], the authors developed a CPU usage monitoring tool for KVM, relying on *"perf kvm record"*. By profiling all CPUs, they could monitor the CPU usage of VMs and the total CPU usage of the hypervisor. In their work, they needed to profile the guest kernel and host kernel at the same time. They were capable of finding the overhead introduced by virtualization for VMs as a whole, but they cannot measure it for each VM separately.

Wang in [106] introduced VMon, monitoring VM interference using perf. From all the available CPU metrics, they used the Last Level Cache (LLC) as an indicator of over commitment of CPU. They showed that LLC has a direct relationship with performance degradation. LLC could be an indicator of CPU over commitment for CPU intensive workloads, but the result will be different when memory intensive tasks are running in the VM. Analyzing nested VMs has been addressed in [107]. They proposed a technique to analyze nested VMs using hypervisor memory forensics. Their tool can analyze nested VM setups and corresponding hypervisors but does not provide any information about nested VMs states and their

execution.

PerfCompass [68] is the only trace-based VM fault detection tool for internal and external faults. It can detect if the fault has a global or local impact. As part of their implementation, they trace each and every VM with LTTng [1]. The data is eventually used to troubleshoot VMs and find out problems like latency in I/O, memory capacity problems and CPU capacity problems. Their approach, however, needs to trace each VM, which significantly increases the overhead on the VMs. Their approach can be ported to nested VMs by tracing each nested VM. Nonetheless, as we will see in subsection 4.7.1, the overhead of tracing nested VMs is much larger than with our proposed method.

The work closest to ours, which motivated the research presented in this paper, is presented in [73]. They proposed a technique to investigate different states of VMs. The authors could find the preempted VMs along with the cause of preemption. In their case, they trace each VM and also the host kernel. After tracing, they synchronize the trace from each VM with that from the host. Then, they search through all threads to find preempted threads. Biancheri *et al.* in [108] extended multi-layer VM analysis. Although this work can be used for nested VMs, the extra efforts required, (tracing the VMs and Nested VMs, synchronizing the traces, finding preempted VMs by searching all available threads in the host and VMs), are all time-consuming.

Early results of this work are presented in [109] [110] [111]. In these papers, we propose a technique to understand the behavior of up to one level of nested VMs. To the best of our knowledge, there is no pre-existing efficient technique to analyze the performance of any level of VMs. Our technique could uncover many issues inside VMs without internal access. Moreover, comparing our method with other possible solutions shows less overhead, and ease of deployment in terms of tracing, since it limits its data collection to host hypervisor level.

4.4 VM and Nested-VM Machine States

Intel-VT (and similarly AMD-V) supports two operating modes, *root mode* and *non-root mode* for executing hypervisor code and VM code, respectively. Furthermore, non-privileged instructions of VMs are executed as non-root mode, and privileged instructions are executed as root mode (at a higher privilege level). The transaction between root mode and non-root mode is called Virtual Machine Extensions (VMX) transition. In each VMX transition, the environment specifications of the VMs and the hypervisor are stored in an in-memory Virtual Machine Control Structure (VMCS) [17].

In the transition between root mode to non-root mode, the state of the hypervisor is saved

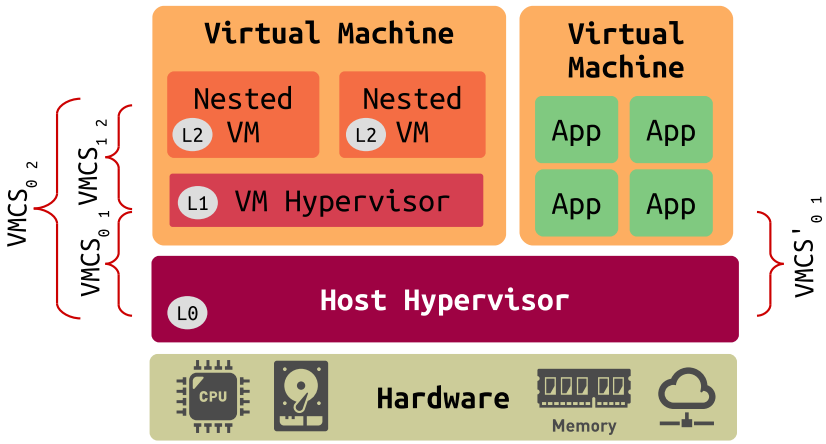


Figure 4.2 Two-Level VMs (Nested VM) Architecture for VMX

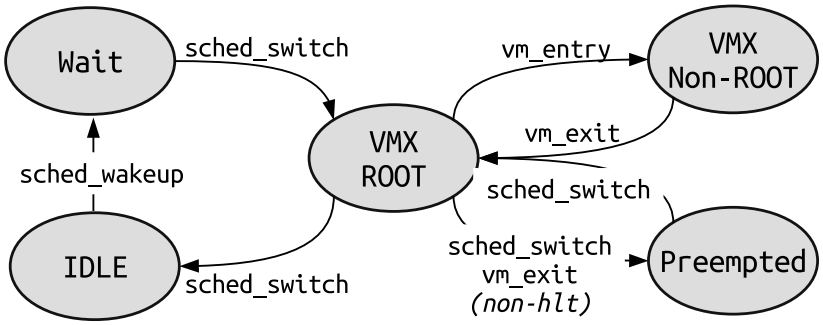


Figure 4.3 Virtual Machine Process State Transition

into VMCS and the environment specifications of the VM are loaded. This is also called a VM entry. On the other hand, in the transition between non-root mode to root mode, the state of the VM is saved into VMCS and the state of the hypervisor is loaded. This is called a VM exit. The Exit reason is a field in the VMCS that changes during a VM exit. It shows the reason for exiting from non-root mode to root mode.

Figure 4.2 shows a two-level architecture for nested VMs. In a two-level architecture, executing any privileged instruction by level two (nested VMs) returns to the host hypervisor (L_0). In this case, the VM hypervisor (L_1) has the illusion of running the code of the nested VM (L_2) directly on the physical CPU. However, privileged instructions of nested VMs should be handled by the highest privileged level. Since L_1 is not the highest privileged level, L_0 handles it. As a result, whenever any hypervisor level or VM executes privileged instructions, the L_0 trap handler is executed. This VMX emulation can go to any level of nesting.

Usually, there is one pointer to a VMCS structure for each vCPU ($VMCS'_{01}$ in Figure 4.2) of each VM. However, for two level of VMs, there are three VMCS structures for each vCPU

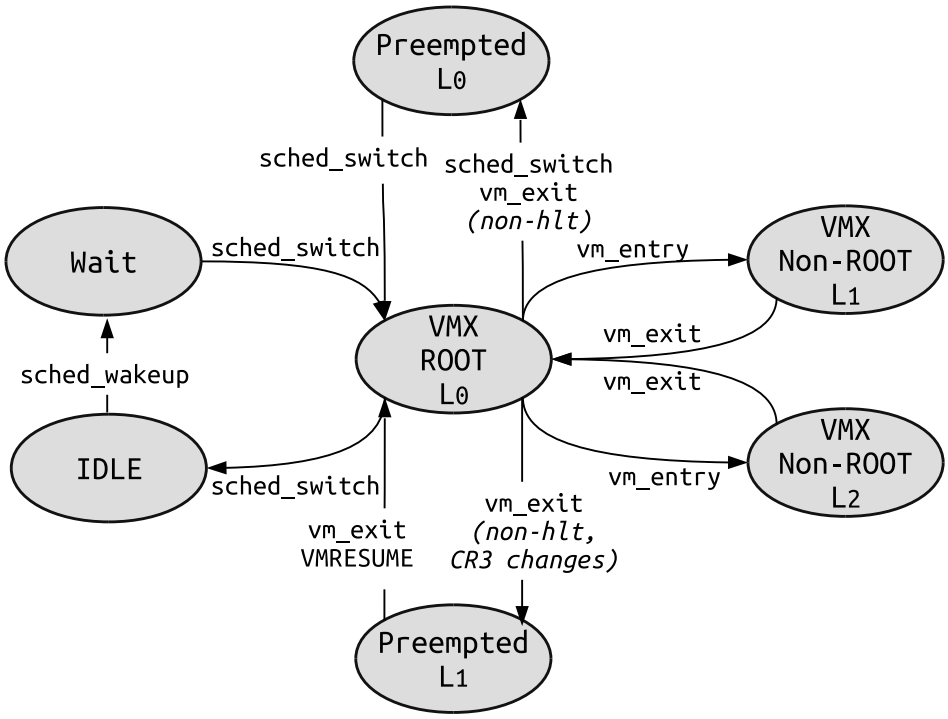


Figure 4.4 Two-Level VMs Process State Transition

($VMCS_{12}$, $VMCS_{02}$, and $VMCS_{01}$ in Figure 4.2). The VM hypervisor uses $VMCS_{12}$ to contain the environment specifications of vCPU of a nested VM. As we mentioned before, the code of a nested VM can be executed directly on the host hypervisor. In this case, the host hypervisor prepares $VMCS_{02}$ to save and to store the state of vCPUs of nested VMs at each VM exit and VM entry. Moreover, the host hypervisor creates $VMCS_{01}$ to execute the code of the VM hypervisor. From the host perspective, $VMCS_{12}$ is not valid (Called shadow VMCS), but the host hypervisor benefits from that to update some fields in $VMCS_{02}$ for each VMX transition. For other vCPUs, there are other VMCS structures that save the environment specifications of the vCPUs.

Figure 4.3 shows different states of a vCPU and the conditions to reach those states. In both root and non-root states, the vCPU is in running mode. In contrast, in the Preempted, Wait, and Idle states, the vCPU is not executing any code. The Preempted state is when the vCPU is being scheduled out by the host CPU scheduler without notifying the guest OS. The Idle state is when the vCPU is being scheduled out voluntarily by sending the *hlt* signal from the guest OS. In the Wait state, the vCPU thread is waiting for the physical CPU for being free to schedule in. Figure 4.4 presents different states of a process inside a two-level VM. In general, a process inside a two-level VM could be in either of these states: host hypervisor as VMX root (known as L_0), VM hypervisor as VMX non-root (known as L_1),

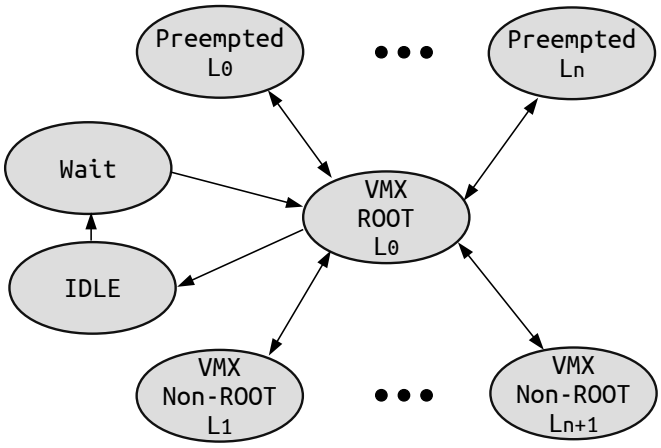


Figure 4.5 n-Levels Nested Virtual Machine Process State Transition

Nested VM as VMX non-root (known as L_2), Preemption in L_1, Preemption in L_0, Wait, and Idle. Executing any privileged instruction causes a VM exit all the way down to the host hypervisor. There are two possible ways of handling any privileged instruction. Along the first handling path, L_0 handles the instruction and forwards it to L_1. In this case, L_0's code is executed in root mode and L_1's code is run in non-root mode. Eventually, L_1 handles the exit reason and launches L_2 in non-root mode. Launching or resuming a VM in L_1 causes an exit to L_0. Then, L_0 handles the exit reason and launches the VM. Along the other possible path, L_0 directly forwards the control to L_2. In this scenario, the exit reason is transparent for L_1, since it happens somewhere else in the host hypervisor level [18]. A process of a nested VM is in the Running mode when it is in the L_0, L_1, or L_2 state. By contrast, the physical CPU is not running code of a nested VM if its state is either Preemption L_0, Preemption L_1, or Waiting state. This can add an unexpected delay to nested VMs, since the nested VM user is not aware of being preempted or waiting for a physical CPU. Figure 4.5 depicts the states of a process inside a n-level nested VM. As shown, a process could be preempted in any level of virtualization, and privileged instructions in the code of Nested VMs could be handled in any level of virtualization. Therefore, new states for preemption in any level, and VMX non-root in any level, are added to process state transitions.

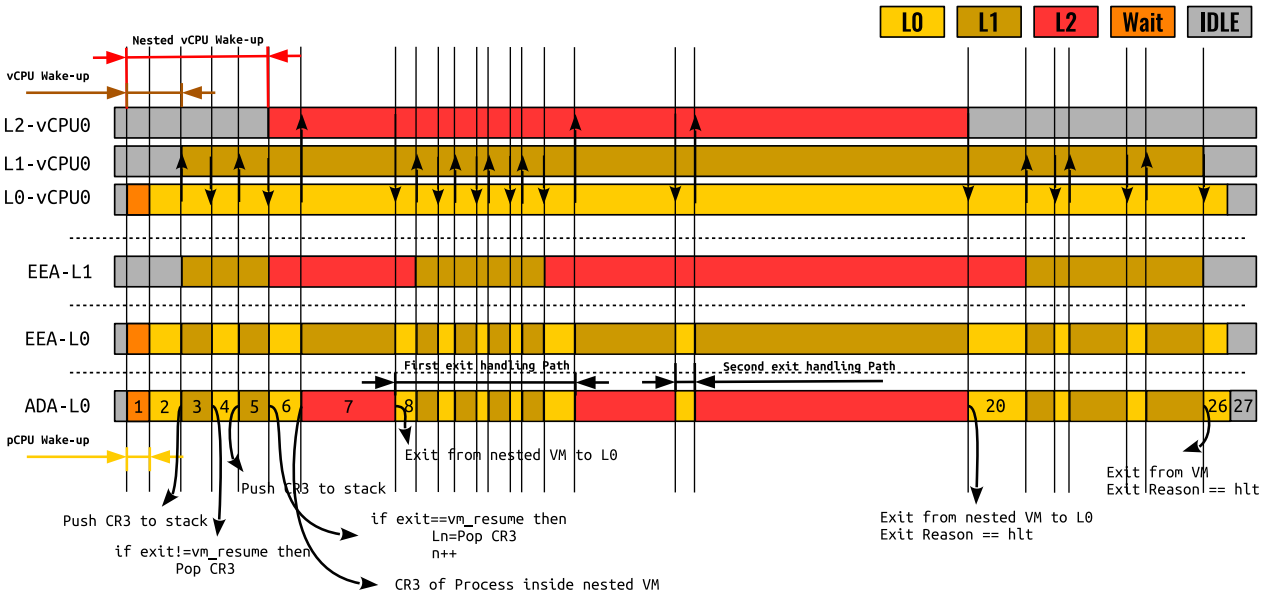


Figure 4.6 vCPU states using different algorithms

4.5 Nested VM Analysis Algorithms

In this section, we propose three algorithms to analyze VMs. **First:** Any-Level VM Detection Algorithm (ADA) to detect nested VMs, at an arbitrary depth from the vCPU thread of a VM. **Second:** Nested VM State Detection (NSD), to uncover the different states of vCPUs for the nested VMs. **Third:** Guest (Any-Level) Thread Analysis (GTA) which is a thread-level and process-level execution time profiling algorithm.

Our algorithms analyze some events that are collected by tracing the host hypervisor. Tracing is a very fast system-wide logging mechanism. An event is generated by the tracer when it encounters an enabled tracepoint at run-time. Events can carry some information like a timestamp and a payload. The payload holds extra information about an event. For example, the payload of a context switch event contains the pCPU and name of the two switched tasks. A timestamp is recorded as time of an event, upon encountering the tracepoint.

4.5.1 Any-Level VM Detection Algorithm (ADA)

In this subsection, we propose an algorithm to detect execution flows for arbitrary depths of nesting from the vCPU thread of a VM in the host. Before introducing Any-Level VM Detection Algorithm (ADA), we explain a very simple VMX root and non-root state detection algorithm called as Entry-Exit Algorithm (EEA). Executing any privileged command in the

Table 4.1 Sequence of events from the host related to Figure 4.6

#	**Event**	**Payload**
1	wake_up	comm=vcpu_thread
2	sched_switch	comm=vcpu_thread, pCPU=0
3	vm_entry	vcpu = 0, CR3 = cr3, SP = sp
4	vm_exit	exit = exit_reason
5	vm_entry	vcpu = 0, CR3 = cr3, SP = sp'
6	vm_exit	exit = vm_resume
7	vm_entry	vcpu = 0, CR3 = cr3', SP = sp"
8	vm_exit	exit = exit_reason
.	.	.
.	.	.
.	.	.
20	vm_exit	exit = hlt
.	.	.
.	.	.
.	.	.
26	vm_exit	exit = hlt

VM (in any level) causes an exit to the hypervisor host (VMX root or L0) which is labeled as `vm_exit` event. After handling the privileged command in VMX root mode, the VM enters the VMX non-root mode which is called as `vm_entry` event. EEA uses `vm_entry` and `vm_exit` events to find out whether the code of the VM is running or the code of the host hypervisor. This algorithm is simplistic and could not detect other virtual levels. Figure 4.6 shows an example of EEA algorithm using the events from L0 (EEA-L0). The states are built using the sequence of events which are shown in Table 4.1. The `wake_up` event shows that vCPU0 is woken up and it goes to the Wait state. The next event is `sched_switch`, which shows that the vCPU0 is being scheduled in and the state changes to L0 (VMX root). When receiving the `vm_entry` event, the state changes to L1 (VMX non-root) and then `vm_exit` changes the state to L0. This algorithm could not reveal if the VMX non-root mode is L1, L2, L3, etc.

In another example, we use the events from the VM in level 1 and then run the EEA algorithm. The result of this example is shown in Figure 4.6 as EEA-L1. As shown, the states are built incorrectly since the exit from L2 goes directly to L0, and L1 does not receive any event.

Using the ADA algorithm, the exact level of code execution can be detected. The ADA algorithm is illustrated with an example in Figure 4.6. When receiving the `sched_switch` event, the vCPU status goes to L0. The `vm_entry` event changes the status to L1 and the CR3 value of the guest is stored as the expected hypervisor CR3. CR3 points to the page

directory of a process in any level of virtualization, and could be used as an unique identifier of a process. The next event, `vm_exit`, modifies the status of vCPU to L0 and, since the exit reason is not vm_resume, it pops the CR3 value from expected hypervisor stack. Upon receiving the 5th event, the vCPU status changes to L1 again. The `vm_exit` with exit reason vm_resume shows that the VM is running a nested VM inside. In this case, the expected hypervisor CR3 is marked as the CR3 of the hypervisor in L1. The next `vm_entry`(Event #7), changes the status of vCPU to L2. As mentioned, executing any privileged instruction in any level of nested VM causes an exit to L0. Therefore, the `vm_exit` modifies the vCPU state to L0. The algorithm uses the marked CR3 to distinguish between L1 and L2.

The pseudocode for the ADA algorithm is depicted in Algorithm 1. The ADA algorithm receives a sequence of events as input and updates the vCPU state. In this algorithm, we uses an array of List of hypervisors for level n. The candidates[m] (m is the level of nesting available) variable is an array of stacks which holds the hypervisor candidate for level n. In case the event is `wake_up`, the state of the vCPU is modified to the Wait state (Line 11). The most important event, `sched_switch`, shows when a vCPU is running on a pCPU. When a vCPU is scheduled in, it goes to the L0 state (Line 15) to load the previous state of the VM from the VMCS. It also updates the lastCR3 hash map to unkown (Line 16) since there is no previous process running on the VM. In contrast, when a vCPU is scheduled out, it goes to either the Idle state (Line 20) or the preempted state (Line 22). The `getLastExit(vCPU)` function returns the last exit for existing vCPU. If the last last_exit is *hlt*, which means that the VM runs the idle thread, the state will be modified as Idle. In other cases of exit reason, it will be changed to the preempted state. When receiving a `vm_entry` event, the state of the vCPU is adjusted to VMX non-root state. First, it uses the `getCandidateLevel()` function to find out the level of entry. The `getCandidateLevel()` function uses CR3, lCR3 (last hypervisor CR3), and a HashMap variable (levels) to find out the level of code execution for CR3. It compares CR3 with the available hypervisor lists in all levels to find out if the CR3 belongs to an hypervisor. In case CR3 is not found in the hypervisor list, it checks lCR3 to find out the last level of code execution (Line 26). The last exit reason is compared if it is VMRESUME or VMLAUNCH (Line 28). If the condition is true, it means that the VM in level n is running another VM. As a result, the last CR3 that was pushed to the hypervisor candidate list will be popped (Line 30), and will be added to hypervisors list in level n (Line 32). If the condition is false, the CR3 will be pushed to the candidate list for level n (Line 38). After calculating the level at the end of this event the state of vCPU will be updated to L_n (Line 39). Receiving any `vm_exit` changes the status to L0 (Line 42). We implemented the proposed algorithm in TraceCompass [2] as a new graphical view for vCPU threads inside the host, and vCPUs inside the VM. The vCPU state is stored in a tree shaped

data based named State History Tree (SHT). In the next section, we present the results of some experiments using the ADA algorithm.

4.5.2 Nested VM State Detection Algorithm (NSD)

As mentioned in the previous section, each vCPU could be in one of VMX root, VMX non-root L_k, Preemption L_k, Wait, or Idle states. Among the aforementioned states, only in the VMX non-root L_n (n is the last level of code execution) state is the actual code of the nested VM being executed directly on a physical CPU. Other states increase completion time of the task inside a nested VM. As a result, finding these states lets the cloud administrator diagnose their VMs and nested VMs better. When a VM or nested VM does not have any code to execute, it exits with the *hlt* exit reason. If and only if a VM or nested VM is scheduled out from a pCPU without exiting with the *hlt* reason, it implies preemption. By observing where this preemption occurs, we are able to distinguish whether it was preempted by the scheduler of the VM or host. For example, in Figure 4.6, event #20 (vm_exit) with exit reason *hlt* shows that the process inside the nested VM does not have any code to run, and the scheduler of the VM runs the idle thread (Event #26). As a result, the scheduler of the host schedules out the vCPU thread from the pCPU.

The Nested VM State Detection (NSD) algorithm is shown in Algorithm 3. The L_n preemption detection happens when the NSD receives the `vm_entry` event. It first inquires whether the CR3 value is a CR3 of the process, and if it was changed or not (Line 14). If the condition is true and the last exit reason is not *hlt*, the vCPU is being preempted by another process in L_n. Using the NSD algorithm, the cloud infrastructure provider could detect if the VM in any level is being preempted, by another VM or process in any level of nested virtualization.

4.5.3 Guest (Any-Level) Thread Analysis (GTA)

As we mentioned in the previous section, in each transition between root mode and non-root mode, the processor state is saved in the VMCS fields. The guest state is stored in the guest-state area in each `vm_exit` and is loaded from the guest-state area at each `vm_entry`. Also, the host state is retrieved at each `vm_exit`. The instruction pointer (IP), stack pointer (SP) and control registers (CR) are some of the registers that are modified during each transition in the VMCS guest-state area.

The process identifier (PID) and process name of each thread inside the guest are not directly accessible from host tracing. The only information which can be uncovered by host tracing about the threads inside the VMs is written in CR3 and SP. CR3 and SP can identify the

Algorithm 1: Any-Level VM Detection Algorithm (ADA)

Input : Trace $\mathcal{T}$
Output: State of $vCPUs$

```
1  ____________________ Initialization ____________________
2  VCPUs ⟵ {inital tid}
3  HashMap levels, lastCR3;
4  List hypervisorsList[m] ;
5  Stack candidates[m];
6  ____________________ Main Procedure ____________________
7  for all event e ∈ T do
8      j = getVMvCPU(e.tid);
9      if e.type is wake_up then
10         if tid == vCPU_j^tid then
11             vCPU_j^State = Wait ;
12     else if e.type is sched_switch then
13         k = getVMvCPU(prev_tid);
14         if next_tid == vCPU_j^tid then
15             vCPU_j^State = L0 ;
16             putLastCR3(vCPU_j^tid,unknown);
17         if prev_tid == vCPU_k^tid then
18             last_exit = getLastExit(vCPU_k^tid) ;
19             if last_exit == hlt then
20                 vCPU_k^State = Idle ;
21             else
22                 vCPU_k^State = Preempted_L0
23     else if e.type is vm_entry then
24         last_exit = getLastExit(vCPU_j^tid) ;
25         lCR3 = getLastCR3(vCPU_j^tid) ;
26         n = getCandidateLevel(CR3,lCR3) ;
27         hypervisors = getHypervisorsList(n);
28         if last_exit == VMRESUME or VMLAUNCH then
29             cHyperviors = getCandidateStack(n);
30             hCR3 = cHyperviors.pop();
31             if !hypervisors.contains(hCR3) then
32                 hypervisors.add(hCR3) ;
33                 putHypervisorList(n,hypervisors) ;
34                 levels.put(hCR3,n);
35                 levels.put(CR3,n+1) ;
36             n ++;
37         if !hypervisors.contains(CR3) then
38             putCandidateStack(n,CR3) ;
39         vCPU_k^State = Ln ;
40     else if e.type is vm_exit then
41         putLastExit(vCPU_j, exit_reason) ;
42         vCPU_j^State = L0 ;
```

Algorithm 2: Utility Functions for ADA Algorithm

```
44 ________ Utilities ________ Function getCandidateLevel(currentCR3, lastCR3):
45 |  int n;
46 |  if lastCR3 == unknown then
47 |  |  n = 0;
48 |  else
49 |  |  n = levels.get(currentCR3);
50 |  return n;
```

Algorithm 3: Nested VM State Detection (NSD) Algorithm

Input : Trace $\mathcal{T}$
Output: State of $vCPUs$

```
1  ____________________________ Main Procedure ____________________________
2  for all event e ∈ T do
3  |  j = getVMvCPU(e.tid);
4  |  if e.type is sched_switch then
5  |  |  else if prev_tid == vCPU_j^tid then
6  |  |  |  last_exit = getLastExit(vCPU_j^tid) ;
7  |  |  |  if last_exit == hlt then
8  |  |  |  |  vCPU_j^State = Idle ;
9  |  |  |  else
10 |  |  |  |  vCPU_j^State = Preempted_L0 ;
11 |  else if e.type is vm_entry then
12 |  |  lCR3 = getLastCR3(vCPU_j^tid) ;
13 |  |  last_exit = getLastExit(vCPU_j^tid) ;
14 |  |  if last_exit != hlt and lCR3 != CR3 then
15 |  |  |  n = levels.get(CR3);
16 |  |  |  vCPU_j^State = Preempted_Ln ;
```

process and thread, respectively. Indeed, CR3 points to the page directory of a process in any level of virtualization. All threads of a process use the same page directory, therefore switching between two threads within the same process does not change the CR3 value. SP points to the stack of the thread inside the VM. As a result, by retrieving these two identifiers, we can find out which thread is executing on a vCPU. To have more information about threads inside the VMs or Nested-VMs, we need to map CR3 and SP to the PID and process name. This is not strictly necessary, since CR3 and SP are unique identifiers of threads, but it is more convenient and human readable if we can map the process info inside the guest with the information we get from the `vm_entry` trace point.

The Guest (Any-Level) Thread Analysis (GTA) is illustrated in Algorithm 7. When the event is `vm_entry`, the stack pointer and CR3 of that thread are gathered from the VMCS guest state, and the process information of the VM is updated. If the mapping information from the VM is available, the CR3 and SP values are converted to the name of the process and threads (Line 3). With `vm_entry` and `vm_exit` events, it queries the state of the vCPU to update the state of the process and thread running on that vCPU (Line 9,14).

Algorithm 4: Guest (Any Level) Thread Analysis (GTA)

Input : Trace $\mathcal{T}$
Output: State of *Processes*

```
1  ____________________ Main Procedure ____________________
2  for all event e ∈ T do
3      j = getVMvCPU(e.tid);
4      if e.type is vm_entry then
5          n = getProcessLevel(CR3,lCR3) ;
6          Status = Query the status of vCPU_j ;
7          Process_n^CR3 = Status ;
8          if Mapping == TRUE then
9              Map SP and CR3 with process memory map of VM ;
10             Change the status of Thread to VMX non-root ;
11     else if e.type is vm_exit then
12         n = getProcessLevel(CR3,lCR3) ;
13         Status = Query the status of vCPU_j ;
14         Process_n^CR3 = Status ;
15         if Mapping == TRUE then
16             Map SP and CR3 with process memory map of VM ;
17             Change the status of Thread to Status ;
```

4.6 Use-Cases

This section covers a variety of known problems for the VMs that our technique could detect. First, the GTA algorithm is being evaluated to profile VM threads and processes. Then, we evaluate our VM tracing technique for the case of detecting issues for different resource types like CPU (subsection 4.6.3), Memory (subsection 4.6.4), and I/O (subsection 4.6.5). Furthermore, different OS types (Linux and Windows) in VM levels are being compared in terms of overhead.

4.6.1 Analysis Architecture

Our approach is independent of the Operating System (OS) and could work on different architectures using Intel or AMD processors. Due to good nesting support, we have chosen KVM under the control of OpenStack, which is the most commonly used hypervisor for Openstack [112]. Our experiments with nested VMs is limited to one level of nesting because of KVM. For the userspace part of the hypervisor, we installed Qemu to execute the OS support for the VM. We also use the same architecture for nested VMs and VM hypervisors. Our architecture is shown in Figure 4.7. As we can see, events are gathered by our tracer (LTTng) from the host hypervisor first, and then the events are sent to the trace analyzer (TraceCompass). Our experimental setup is described in Table 5.2. The Qemu version is 2.5 and the KVM module is based on Linux kernel 4.2.0-27.

Table 4.2 Experimental Environment of Host, Guest, and NestedVM

	Host Environment	**Guest Environment**	**Nested VM Environment**
CPU	Intel(R) i7-4790 CPU @ 3.60GHz	Two vCPUs	Two vCPUs
Memory	Kingston DDR3-1600 MHz, 32GB	3 GB	1 GB
OS	Ubuntu 15.10 (Kernel 4.2.0-27)	Kernel 4.2.0-27	Kernel 4.2.0-27
Qemu	v2.5	v2.5	-
LTTng	v2.8	v2.8	v2.8

The analysis is performed according to the following steps :

1. Start tracing on the host;
2. Run the VMs of interest;
3. Stop tracing;

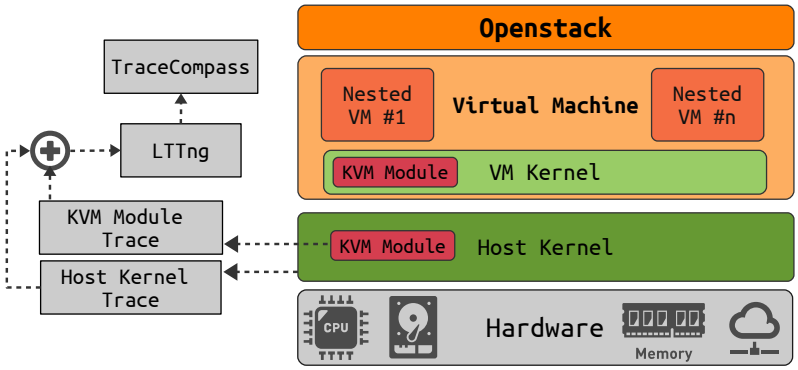


Figure 4.7 Architecture of our implementation

4. Run the analyses;

5. Display the result in the interactive viewer;

Among the available Linux tracers, we choose a lightweight tracing tool called the Linux Tracing Toolkit Next Generation (LTTng) [1] due to its low overhead kernel and userspace tracing facilities. Furthermore, in Linux, the KVM module is instrumented with static tracepoints and LTTng has appropriate kernel modules to collect them. Therefore, LTTng is particularly suitable for our experiment, since it collects Linux kernels and KVM module events with a low impact on VMs. We also added our own tracepoint (vcpu_enter_guest) to retrieve the CR3 and SP values from the VMCS structure, on each VMX transition, using kprobe. After the relevant events are generated and collected by LTTng, we study those with the trace analyzer, as elaborated in the next subsection. The events required for the analysis and the instrumentation method, along with their name in LTTng, are shown in Table 5.3.

We implemented our event analyzers as separate modules in TraceCompass [2]. TraceCompass is an Open-source software for analyzing traces and logs. It provides an extensible framework to extract metrics, and to build views and graphs.

Table 4.3 Events required for analysis

Category	Event	LTTng Event	Method
scheduler	sched_switch	sched_switch	tracepoint
scheduler	wake_up	sched_wakeup	tracepoint
hypervisor	vm_exit	kvm_exit	tracepoint
hypervisor	vm_entry	kvm_entry	tracepoint
hypervisor		vcpu_enter_guest	kprobe

4.6.2 Thread-level and Process-level Execution Time Profiling

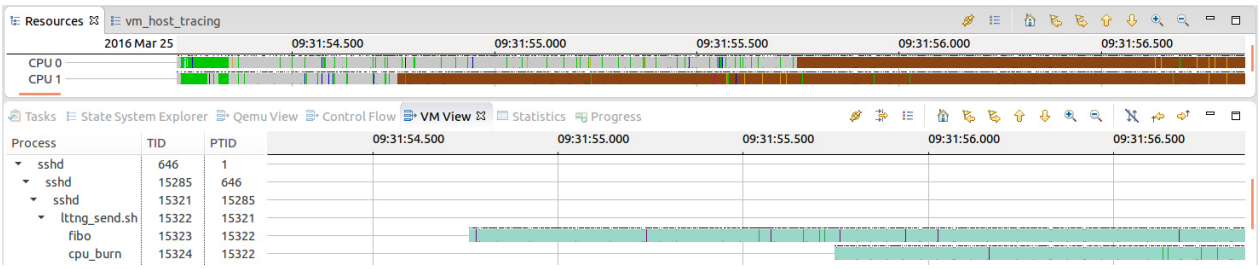


Figure 4.8 Control flow view of threads inside the virtual machine

We implemented the GTA algorithm in TraceCompass as a new graphical view for threads inside the VM. To examine our algorithm, we wrote two C programs that calculate Fibonacci numbers in a busy loop. We named them Fibo and cpu_burn. We used ssh to connect to the VM. The Fibo program is run and, after 1 second, the cpu_burn program is run. Figure 4.8 shows the resource view of the host and the VM thread view at the same time. We see that first a vCPU thread runs on pCPU1 and, after 1 second, another vCPU thread executes on pCPU0. We traced the host hypervisor and we used the GTA algorithm. The VM Thread view displays the threads running on these two pCPUs at that time. We observe that the Fibo program was running on pCPU1 and cpu_burn was executing on pCPU0. As this result shows, the GTA algorithm could profile a process in a VM or Nested VM, without internal access.

The analysis of our tool reveals that just enabling the `sched_switch` tracepoint inside the guest adds almost 3% overhead to the guest execution. We claim that our approach adds a much lower, mostly negligible, overhead to the VM when the process information is dumped once during the whole trace.

4.6.3 CPU Cap and CPU Overcommitment Problem

Predicting VM workloads is a big challenge. Sometimes, cloud users set the VM CPU cap too low, which causes inadequate CPU allocation. In another case, the cloud administrator overcommits CPU resources and shares among too many VMs. In both cases, these problems cause latency for the VMs. In our first experiment (S1-DiffGrp), we use **Hadoop** to calculate the nth binary digit of **pi** by using the Bailey Borwein Plouffe (BBP) formula. Any digit of pi can be calculated by BBP without calculating prior numbers. Therefore, calculating pi could be split into different tasks and could be mapped to several nodes. We conducted

our experiment on a cluster where each VM had 2 vCPUs with 3 GB of memory. One VM is designated as master and 3 VMs are configured as Slave. All VMs are assigned to different resource groups in order to reduce the interference between VMs. In our experiment, each resource group had 2 physical CPUs (pCPU) and 6 GB of memory. The Hadoop version was 2.7 and Java version was 8. We used the YARN framework for job scheduling and resource management, and the Docker Container Executor (DCE) to allow the YARN NodeManager launching YARN containers and running user code. During our experiment, the Hadoop resource manager created 8 containers on each slave node and the Hadoop application manager submitted 8 tasks to each of them. A job to calculate 500 digits of pi is submitted and is split into 24 maps. We realized that sometimes the execution time for calculating pi is more than expected. We investigated further and found out that node VM-Slave 2 finishes its associated task after other nodes (the application manager was in node VM-Slave 4). The same job is submitted again and the host is traced with LTTng. In our investigation with LTTng and NSD (shown in Figure 4.9), we found that node VM-Slave 2 is using more CPU compared to other VM-slaves. VM-Slave 2 is compared with VM-Slave 3 by looking at their processes (GTA Algorithm). Based on the comparison, VM-Slave 3 had 8 active processes, but VM-Slave 2 had 9 active processes. Comparing processes of VM-Slave 2 and VM-Slave 3 shows that process with CR3 5263425536 is not a Hadoop container and uses one CPU most of the time. This process could be another application that is scheduled to run at a specific time (e.g., update). In this scenario, Hadoop containers should share vCPU with another application inside the VM-Slave 2, which is the cause for the delay in finishing associated task.

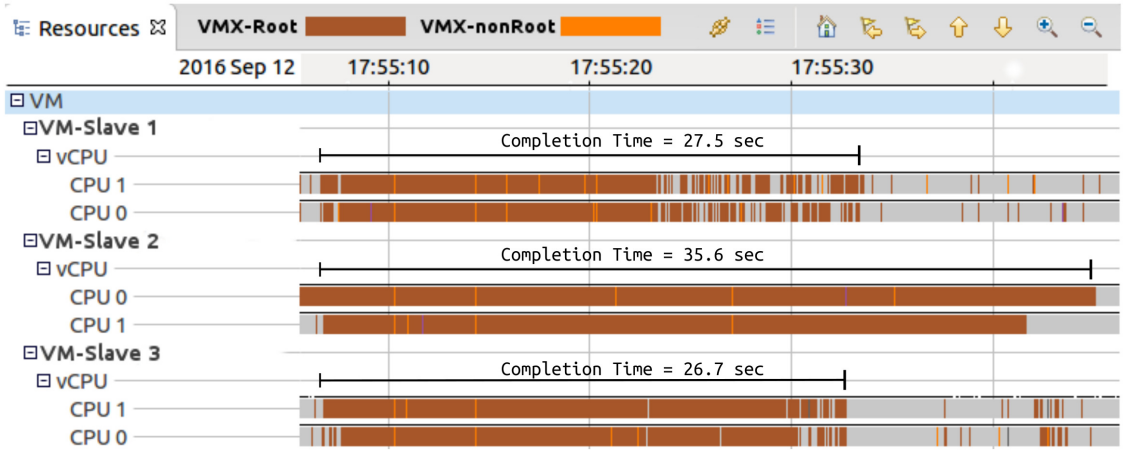


Figure 4.9 Three slaves running one submitted task- VM-Slave 2 responses late to each task

In the second experiment (S2-SameGrp), we used the same configuration except that we put

two VM-Slaves (VM-Slave 1 and VM-Slave 2) in the same resource group. We observed that the tasks submitted to VM-Slave 1 and VM-Slave 2 took more time to be completed (around 49% increase). The results of S2-DiffGrp and S2-SameGrp are shown in Table 4.4. We investigated the reason by using the NSD and found that VM-Slave 1 and VM-Slave 2 are preempting each other most of the time. The reason is that the tasks submitted to both VM-Slaves were a CPU intensive job and VMs fought for existing CPUs.

Table 4.4 Completion time for Hadoop VM-Slaves in different scenarios.

Scenario	*VM name*	*Completion Time (sec)*	*Usage Time (sec)*	*Preemption Time (sec)*
S2-DiffGrp	**VM-Slave 1**	27.5	46.832	0.102
	VM-Slave 2	35.6	70.795	0.104
	VM-Slave 3	26.7	49.144	0.238
S2-SameGrp	**VM-Slave 1**	53.917	55.971	13.89
	VM-Slave 2	53.328	53.532	13.47
	VM-Slave 3	27.904	54.756	0.141

As discussed, preemption could happen in any level of virtualization, which is a cause of latency. In this new scenario, we show how our analysis could reveal unexpected delays in nested VMs. For these experiments (S3-Nest1VM), we configure our testbed as explained in section 4.6.1. Sysbench is set to run 60 times and compute the first 1000 prime numbers. After each task execution, it waits for 600 ms and then re-executes the task. We start a VM with two vCPUs and a nested VM with two vCPUs inside. We pin the nested VM vCPUs to the vCPU 0 of the VM and we pin the vCPUs of the VM to the pCPU 0 of the host. We do this to ensure that the code of nested VMs executes on pCPU 0. As expected, the execution time for the same task should be almost equal. On average, the completion time for finding the first 1000 prime numbers is 327 ms, with a standard deviation of 8 ms.

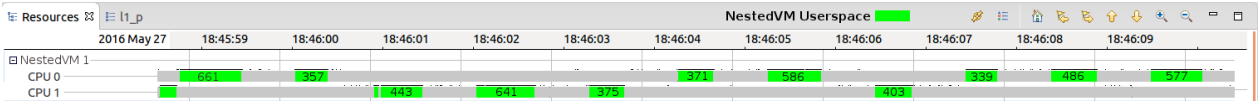


Figure 4.10 Execution time of the prime thread (CPU view)

In the next experiment (S4-Nest2VM), we launch two nested VMs in VM testU1. Both nested VMs have two vCPUs that are pinned to vCPU 0 and vCPU 1 of the VM. The rest of the configuration is kept the same as in the previous experiment, with the exception of

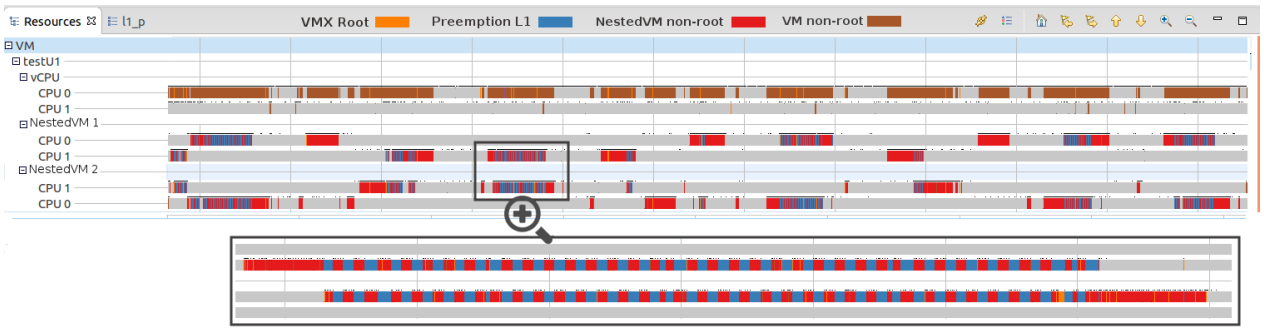


Figure 4.11 Resource view of CPU for two nested VMs inside VM testU1 by host tracing - L1 Level Preemption

Sysbench executing in the nested VM2, being configured to wait 1 sec after each execution. We start Sysbench at the same time for both nested VMs and we start tracing the nested VM1 with LTTng. In our investigation with LTTng, we realized that the execution time for the same task varied more than expected. Figure 4.10 shows the execution time for the same load. We see that it varies between 339 and 661 ms. The execution time for 60 executions of the same load is 465 ms with a standard deviation of 120 ms. To investigate the cause of the execution time variation, we traced the host and used our NSD algorithm to detect the different states of nested VMs. Figure 4.11 shows the result of our analysis as a graphical view. By tracing only the host, we first detect that the testU1 VM is running two nested VMs. Then, we further find out when the code of each nested VM is running on the physical CPU. By looking at the view, we can infer that, during the execution, two nested VMs are preempting each other several times. For more details, we zoom in a section where the two nested VMs are preempting each other, and can observe the events along with fine-grained timing. This preemption occurs at the VM hypervisor level and is more or less imperceptible by the host hypervisor.

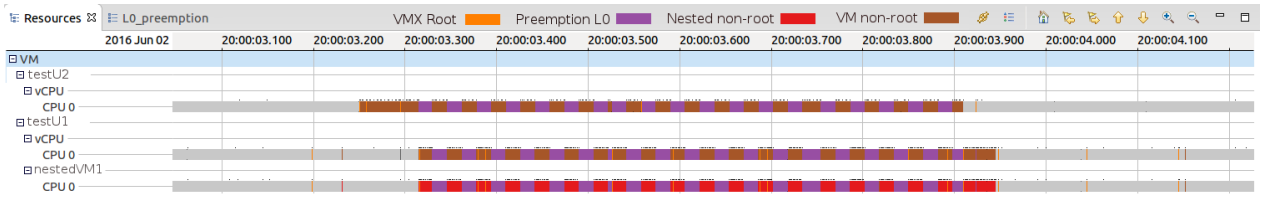


Figure 4.12 Resource view of CPU for one nested VM inside VM testU1 preempted by testU2 by host tracing - L0 Level Preemption

In another experiment (S5-2VM1NestVM), we turn off one of the nested VMs and launch

two other VMs in the host. The VMs and Nested VM are configured as before, except that now we set Sysbench to wait 800 ms after each execution in the VMs. Our investigation shows that the completion time on average for 60 runs of the same load on the nested VM is 453 ms, with a standard deviation of 125 ms. We traced the host hypervisor and exploit our NSD algorithm to investigate the problem further. As Figure 4.12 shows, the nested VM inside VM testU1 is being preempted. In this experiment, the preemption occurs at the host hypervisor level, when VMs are preempting each other.

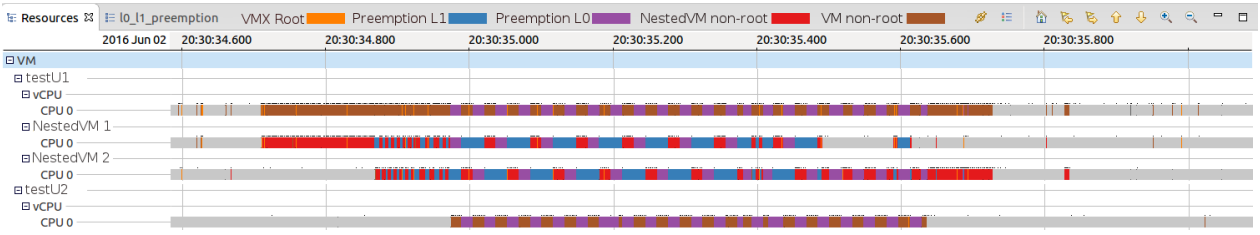


Figure 4.13 Resource view of CPU for two different nested VMs inside VM testU1 preempted by VM testU2 and each other by host tracing - L0 and L1 Levels Preemption

In the next experiment (S6-2VM2NestVM), we launch another nested VM inside VM testU1 (NestedVM 2). We also start VM testU2 and set Sysbench to find the first 1000 prime numbers, like in the previous experiment. In this experiment, each VM and nested VM have one CPU and all CPUs are pinned to pCPU 0. We start the test at the same time for the VM and all nested VMs. As a result of this experiment, we find that the completion time for the same task varies a lot. On average, the execution time for each task takes 651 ms, compared to 327 ms in the first experiment. Moreover, the standard deviation for 60 Sysbench runs was 371 ms. We investigated the cause of this problem by executing the NSD algorithm. Figure 4.13 shows that nested VMs were preempting each other along with VM testU2. In this test, we have preemptions from L_0 and L_1, which cause serious delays in the completion time of tasks. It is worth mentioning that none of these observed preemptions, at any level, are detectable with conventional state-of-the-art tools.

There is a trade-off between CPU utilization and preemption. As CPU utilization increases, more preemptions occur. IaaS providers wish to increase resources utilization to gain profit while maintaining a high QoS to stay in the business. Overcommitment of CPUs may cause serious latencies for VMs. Therefore, preemption can be one of the most important factors in the service level agreement (SLA) between the VM user and the Cloud provider. Using our analysis, the cloud provider could find out when preemption occurs and which VM is preempting others more.

4.6.4 Memory Overcommitment Problem

In this section, we represent how overcommitting the memory could increase the latency in VMs. Real-time information delivery, which uses in-memory storage, is an emerging topic that comes with cloud computing technology. In-memory databases are faster than disk-optimized databases, since access to memory is faster than disk. MemSQL, Redis, and Memcached are examples of in-memory data storage. Github, Twitter, and Flickr use different in-memory storage applications on top of cloud infrastructures to deliver real-time information to their users [113]. In this experiment, we show the ability of our technique to detect memory problems. Each `vm_exit` event has a reason, which is written in the exit_reason field. For example, if a `syscall_read` executes in the VM, it causes a `vm_exit` with exit reason of 30, which is I/O instruction [17]. The frequency of each different exit reason contains a lot of information about the instructions running in the VM. For example, a high frequency of exit reason 30 shows intense I/O activity in a VM.
EPT violation is another `vm_exit` reason that changes the state of the vCPU from VMX non-root to root. It occurs when a VM attempts to access a page that is not allowed by the EPT paging structure, known as a VM page fault. IaaS providers overcommit virtual resources to maximize utilisation and thus use fewer servers. However, sometimes overcommitting virtual resources saturates the resources and causes some issues for VMs. In order to find out frequent exit reasons, we wrote an analysis that could determine the more frequent exit reasons and the associated execution duration. This analysis can help us to guess the behavior of the thread running in the VM and uncover any undue latency.

The same architecture as described in section 4.6.3 is being used for our new experiment. We wrote a C program that writes random numbers into 1GB of memory, named *eat_mem*. This program executes frequently in VM1, VM2, and VM3. We also wrote another program that randomly executes a small CPU intensive task inside VM4 and VM5. Furthermore, in order to overcommit the memory, we modified the eat_mem program to use 25 GB of RAM in the host. The result of our experiment is found in Table 5.5. We observed that VM1, VM2, and VM3 suffered more from overcommitting the memory since they were executing a memory intensive program. VM1, VM2, and VM3 were executing eat_mem for 1.5s in average, but 15% of their time is wasted in average, because of overcommitting the memory. Also, we can infer that VM1 is suffering more from memory overcommitment. Our technique is also able to detect memory overcommitment inside a VM. We could find out memory overcommitment over any level of virtualization by using the NSD algorithm.

Table 4.5 Execution time for different VMs when the host is suffering from Memory overcommitment.

VM name	*Execution Time(ms)*	*Freq EPT Violation*	*Violation EPT Time(ms)*	*Percentage(%)*
VM1	1329.0	3554	237.4	17.8
VM2	1834.5	18801	260.5	14.2
VM3	1332.4	15288	141.2	10.6
VM4	1169.1	0	0	0
VM5	1857.8	30	0.2	0

4.6.5 Overhead of Virtualization Layer for Different Types of Workload

In this section, the latency added to the applications inside a VM in any level is discussed. In addition, two possible ways of handling any privileged instruction are studied. For the first handling path, where L_0 handles the instruction and forwards it to L_1, we wrote an application to read 32 sectors of the disk.

This type of workload is I/O intensive and needs numerous interactions between the different virtualization layers. For another possible path, where L_0 handles the privileged instruction and then directly executes the VM code, we used our Fibo application to calculate 10000 Fibonacci numbers. We execute these applications inside a VM and a Nested VM. Figure 4.14 depicts the percentage of the elapsed time in the different layers of virtualization. When a VM or a nested VM runs a CPU-intensive job, the percentage of application code executing is much higher than handling privileged instructions (around 99%). As can be inferred from this figure, in this case, the nested VM rarely exits to L_0 and updates L_1. In contrast, the VM exits to L_0 and then enters L_1 to update the necessary information when it is executing an I/O intensive job. In average, in our experiment for a nested VM, our application runs 26.31% of the time, and otherwise executes code of different levels of the hypervisor (around 73%).

In another experiment, an RPC server and client are written to experiment the effect of wake-up latency on applications inside VMs and nested VMs. In this experiment, our RPC server could accept any command from the RPC client and executes it. The RPC client sends a sleep(0.1) command to the RPC server to execute every 100 ms. This causes our RPC server to run frequently. As Figure 4.15 shows, for a nested VM the added overhead is higher than for a VM without nesting. The reason is that different layers of virtualization need to be updated.

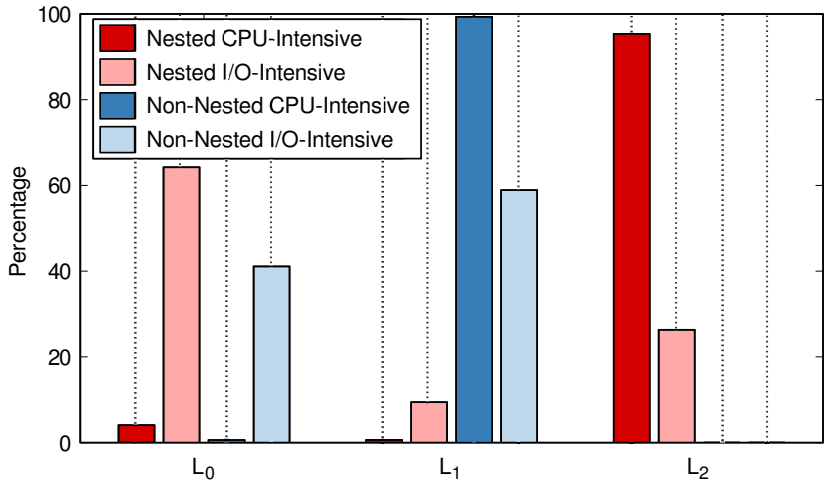


Figure 4.14 Overhead of Virtualization for different types of workload

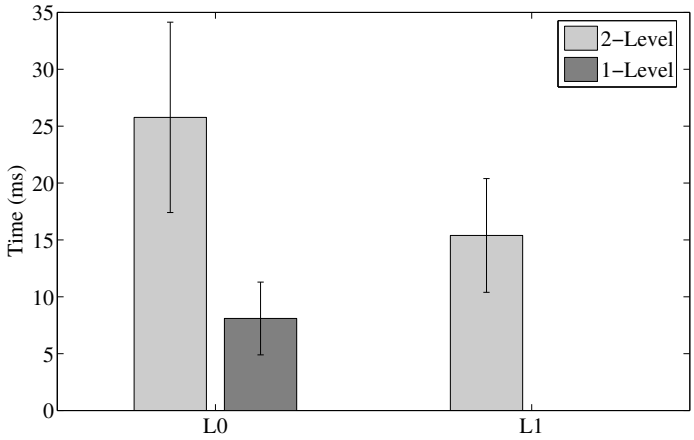


Figure 4.15 Average wake up latency for Nested VMs and VMs

4.6.6 Overhead of Virtualization for Different Operating Systems

In this subsection, we present the results of virtualization overhead imposed by different levels of virtualization, when the OS in the VM is not Linux. The added overhead, imposed by the different layers of virtualization is,

$$O_{VM_j} = \sum_{i=0}^{n-1} T_{L_i} \tag{4.1}$$

Where T_{L_i} is the elapsed time in different levels of hypervisors. The utilisation rate of the application running inside the VM is calculated in Equation 4.2. It is defined as the time

when the actual code of the VM is running divided by total time of running an application on the pCPU.

$$U_{VM_j} = \frac{T_{L_n}}{\sum_{i=0}^{n} T_{L_i}} \tag{4.2}$$

In this experiment, we used Qemu-KVM as hypervisor. In Qemu-KVM, two levels of nested virtualization are implemented and the main thread of Qemu-KVM is responsible for reacting to events that are dispatched to event handlers. If the received event is an I/O event, the task will be submitted to worker threads. Otherwise, the main thread wakes up the vCPU thread and injects the event as an interrupt to the vCPU. We tested our method for a simple C program, named as Fibo, that calculates 10000 Fibonacci numbers. We executed our Fibo program on one level of virtualization and two levels of virtualization. Furthermore, to show that our method works for any VM OS, we tested the Fibo program on the Windows and Linux OS. We observed that running the Fibo program on Windows takes more time than on Linux, for both one level and two levels of virtualization. We investigated this further with our thread analyzer and found out that a specific thread named System is running periodically inside Windows. The System process inside Windows OS is responsible for handling interrupts. We also discovered that the main thread of Qemu-KVM injects interrupts periodically to the VM, and the System process inside Windows handles it. Further investigation showed that the injected IRQ is the timer interrupt which we did not observe when using Linux. Newer Linux VM on the Qemu-KVM hypervisor uses the kvm-clock as paravirtual clock device. When the guest starts, it creates a memory page that shares the kvm-clock data with the hypervisor. The hypervisor constantly updates the clock data with the time information. Therefore, the guest does not need to use a local APIC timer interrupt on each CPU to generate the scheduler interrupt. Contrary to Linux VM, Windows VM on KVM uses periodic RTC clocking that requires interrupt rescheduling for time keeping. When running Windows, this paravirtualization optimisation is not programmed in Windows, and even if the VM is idle, the main thread of Qemu-KVM injects timer interrupt almost every 15.6 ms. Otherwise, when a program is running, the main thread of Qemu-KVM injects the timer interrupt more frequently. As a result, more computing power will be wasted. We did this experiment 10 times and the average of our results is depicted in Table 4.6. In the case of Nested Windows, the VM frequently exits from L_2 to L_1, and then, to resume the nested VM, it goes to L_1 to update $VMCS_{12}$. The added overhead is mainly because of the injected timer update interrupt. We can see that the utilization rate decreases from 98.5% to 69.5% when we use a nested Windows VM. Using nested VMs with Linux for CPU intensive tasks does not add much overhead to the execution of our jobs. As we observed, the nested VM

adds 0.1% overhead to the execution of our Fibo program.

Table 4.6 Execution time for Fibo program on different levels of virtualization and different Operating Systems.

Experiment	T_{L_0} *(ms)*	T_{L_1} *(ms)*	T_{L_2} *(ms)*	U *(%)*	O *(ms)*
Nested-Linux	18.779	4.728	1539.45	98.5	23.507
Nested-Windows	439.864	283.582	1653.62	69.5	723.446
VM-Linux	5.623	1512.18	-	99.6	5.623
VM-Windows	216.362	1569.1	-	88.1	216.362
Host	1508.75	-	-	1	-

4.7 Evaluation

4.7.1 virtFlow Overhead Analysis

In this subsection, we compare two other existing approaches with the NSD algorithm in terms of added overhead to the nested VMs.

The first approach is to trace the host and guest hypervisors (L_1L_0) and then use the method that is proposed in [109]. Another technique is to trace both hypervisors and each nested VM ($L_2L_1L_0$) [108]. In both approaches, the cloud administrator needs the authorization to access each VM and Nested VM. Table 6.4 presents the added overhead to the nested VMs for the different algorithms. We configured the Sysbench benchmark to study the overhead by running 60 times CPU, Disk I/O, and Memory intensive evaluations. To evaluate the network overhead, *iperf* is being configured. Then, we averaged all results, to avoid unexpected latencies in our analysis. We enabled all the necessary events for each analysis. It is worth mentioning that other approaches need to access VMs and nested VMs, as compared to our new proposed approach which is purely a host hypervisor-based algorithm. As shown in the table, our approach adds less overhead to the nested VMs, since it only traces the host hypervisor. In the CPU, Memory, and Network intensive workloads, we add negligible overhead. For the I/O intensive evaluation, the overhead is 34.6 %, which is expected since LTTng is also using the same Disk to store the trace. Indeed, the performance of a disk degrades significantly when two processes compete to access the disk, since each may have an efficient sequential access load, but the mix of the two becomes an inefficient seemingly random access load. This is a well-known problem and using a separate disk for storing the trace data is recommended whenever I/O bound processes are being traced.

Table 4.7 Comparison of our approach and the other multi-level tracing approaches in term of overhead for synthetic loads

Benchmark	*Baseline*	$L_2L_1L_0$	L_1L_0	*NSD*	*Overhead(%)*		
					$L_2L_1L_0$	L_1L_0	NSD
File I/O (ms)	546	809	773	735	48.2	41.5	34.6
Net I/O (GB)	50.2	12.7	33.3	48.2	28.65	6.72	3.98
Memory (ms)	497	505	503	502	1.6	1.2	1
CPU (ms)	334	351	340	339	4.9	1.8	1.4

4.7.2 Ease of Deployment

Our technique for analyzing VMs uses a few (four events) host hypervisor tracepoints. Other available methods ([108] and [73]), trace each relevant VM and the host. In the absence of a global clock, between the host and each VM, a synchronization method must be used. A big challenge of synchronization methods is that they need extra events in order to be sufficiently precise. This adds extra overhead to the VMs and host. It also increases the completion time for the analysis part. Our method does not need any synchronization, since it receives all the events from the same clock source in the host. It could be implemented on other OS types, since it only needs to enable the events illustrated in Table 5.3.

4.7.3 Limitations

Our technique is limited to the host data, and guest OS specific information is not accessible with our method. For example, our technique could not detect a container in a VM, but it could show it as a separate process using the GTA algorithm. In contrast, other trace-based methods ([108] and [73]) provide more useful insights about running processes and their interaction with the guest kernel.

4.8 Conclusion

Nested virtualization is frequently used for software scaling, compatibility, and security in industry. However, in the nested virtualization context, current monitoring and analysis tools do not provide enough information about VMs for effective debugging and troubleshooting. In this paper, we address the issue of efficiently analyzing the behavior of such VMs. Our technique can detect different problems along with their root causes in nested VMs and their corresponding VMs. Furthermore, our approach can uncover different levels of code execution among all the host and nested VMs layers. Our approach is based exclusively on

host hypervisor tracing, which adds less overhead as compared to other approaches. Our benchmarks show that the added overhead in our approach was around 1%. In contrast, the overhead of other approaches ranged from 1.2 to 4.9%. We also proposed a way to effectively visualize the different levels of code execution in nested VMs along with their state. These graphical views show the timing at high-resolution of all VMs and nested VMs executions. Our technique is being tested for different types of guest OS to investigate performance issues. As future work, our current technique can be enhanced to further investigate interferences between VM and nested VMs. It could be used to understand the cause of waiting for each process inside the VM. In addition, based on the extracted metrics, we could group VMs and Nested VMs based on their behavior.

CHAPTER 5 ARTICLE 2: "CRITICAL PATH ANALYSIS THROUGH HIERARCHICAL DISTRIBUTED VIRTUALIZED ENVIRONMENTS USING HOST KERNEL TRACING"

Authors

Hani Nemati, Francois Tetreault, Jason Puncher, and Michel Dagenais

Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

Submitted in IEEE Transactions on Cloud Computing

Reference as H. Nemati and M. R. Dagenais, "Critical Path Analysis through Hierarchical Distributed Virtualized Environments using Host Kernel Tracing," Under-Review

5.1 Abstract

The dynamic nature of applications in Virtual Machines (VMs) and the increasing demand for virtualized systems make the analysis of dynamic environments critical to achieve efficient operation of such complex distributed systems. In this paper, we propose a precise host-based tracing and analysis method to retrieve execution flows, and dependency flows from virtualized environments, regardless of the level of nested virtualization. Given a host operating system level trace, the Any-Level vCPU Detection (ASD) algorithm and Guest Thread-state Analysis (GTA) algorithm detect the different states of vCPUs and threads for arbitrary nesting depths. Then, the Execution-graph Construction (HEC) algorithm extracts the waiting / wake-up dependencies chains out of the running processes across VMs, for any level of virtualization in a transparent manner. The process dependency graph, vCPU state, and VM process state are displayed in an interactive trace viewer, Trace Compass, for further inspection. Our proposed VM trace analysis algorithms have been open-sourced for further enhancements and collaborative research and development. Our new techniques were evaluated with workloads generated using several well-known server applications (e.g., Hadoop, Apache, MySQL, Linux apt-get, and IMS network). The proposed approaches are based on host hypervisor tracing, which brings a lower tracing overhead (around 1%), is easier to deploy, and presents fewer security issues as compared to other approaches.

5.2 Introduction

Cloud computing has considerably changed the computing paradigm. Many enterprises are migrating their services onto cloud platforms to benefit from resource pooling, resource provisioning, and the Pay-per-Use model. Despite the well-known advantages of the cloud platform, there are nonetheless important performance challenges due to the different virtual layers and the variety of applications competing for shared physical resources. As the number of software and hardware components increases, identifying a performance problem becomes potentially difficult in this context. For such complex layered systems, a comprehensive method is needed to understand what goes on inside. Moreover, the IaaS providers may allow the cloud users to run VMs that run their own hypervisor (Virtual Machine Monitor or VMM) with VMs, creating nested VMs. In this case, the diagnosis of performance issues is even more complex.

While it is possible to get a comprehensive picture of the layered system execution, by tracing at each and every layer, this is generally not possible nor desirable. Indeed, the cloud provider usually does not have access to the client VMs in order to install a tracing agent. Furthermore, tracing at each and every layer generates a lot of redundant data and incurs a much larger overhead that host-only tracing.

Agent-less VM tracing, collecting trace data only at the host hypervisor level, also enables additional use cases. Some examples are: **1)** The VMs kernel is too old to support modern tracing tools. **2)** The VMs Kernel is closed-source and there is no tracer available. **3)** The VMs do not have sufficient resources to run the tracer properly.

Although the cloud environment resources are shared among VMs, the applications within a VM have the illusion of having the available resources to themselves. Correspondingly, an application could be tested and validated in an environment but could fail in the virtualized environment. Thus, it is important to understand and quantify the physical resources usage of each VM, even though the application behaviour may change over time. Tracking these changes would help IaaS providers in tuning the resources allocations among VMs. The proposed solution aims at supporting the IaaS provider in understanding the resources consumption, and the dependencies among processes, even across VMs.

The best way investigate the processes dependencies is to analyze the wait and running states of the VM processes. This allows to detect why and when a process waits. A process typically waits in one of the four following scenarios. When the process is woken up by a timer interrupt, it indicates that a timer fired inside the VM. If the process is woken up by another process, it indicates that it was waiting for another process to finish its task.

When a process is woken up by network interrupts, it shows that the VM was waiting for an incoming packet. Finally, the VM could also wait for a block device like a disk, in which case the process is resumed by a disk interrupt.

In this paper we propose, implement, and evaluate the Host-based Execution-graph Construction (HEC) algorithm to determine the dependencies among the processes and the resources. The active path is presented through an interactive viewer for further inspections. We also propose two techniques, Any-Level vCPU States Detection Algorithm (ASD) and Guest (Any Level) Thread-state Analysis (GTA), to deduce for VMs and nested VMs the state (and reason for waiting) of their vCPUs, processes, and threads. Using our method, public cloud providers like Amazon EC2, Microsoft Azure and Rackspace can detect processes and extract information from running VMs. Our method does not require any agent located in the VM to extract information about processes, threads and their dependencies. Moreover, the entire analysis of the collected data is done completely at the host level. HEC may be used not only to diagnose application performance problems, but it can also be exploited as a reverse engineering tool. Indeed, it is able to find out the execution pattern of an arbitrary process inside a VM. It unveils the process resources usage along with its interactions with other contending processes. The analysis is independent of program type, VM operating system, and the virtualization layers. It also imposes a very low-overhead to the running application, as compared to existing tools. To the best of our knowledge, there is no pre-existing efficient technique to analyze the performance of VMs and nested VMs and to construct the process dependency graph across a distributed environments with nested virtualization layers.

Our main contributions in this paper are: **First**, we propose a fine-grained VM vCPU and nested VM vCPU state analysis (including reasons for blocking) based on host hypervisor tracing. All the tracing and analysis phases are hidden from the VMs. **Secondly**, we propose a method to detect the different states of processes and threads, not only inside the VMs but also inside nested VMs. **Thirdly**, we propose a VM and Nested VM critical path analysis technique, to follow execution path of arbitrary processes, over the network as well as through multiple virtualization layers. **Fourthly**, we have implemented a graphical view for processes inside VMs. Our graphical view presents a timeline for each individual process, with different states along with other interactive processes. **Fifthly**, we demonstrated the power of the proposed technique through different representative use cases based on well-known applications like Apache, MySQL, Hadoop, Linux Advanced Packaging Tool (APT) and IP Multimedia Subsystem (IMS) network and actual use cases inspired from industry.

The remainder of this paper is structured as follows. Section 5.3 reviews the related work.

Section 5.4 explains some background information about virtual interrupt injection, along with different available states for the VM processes. In section 5.5, we present the algorithm used to detect the different states of processes and their dependencies. Section 5.6 demonstrates our experimental results. We also compare our method with other available methods in terms of overhead in sub-section 5.7.2. Section 5.8 concludes the paper with directions for future investigations.

5.3 Related Work

Obtaining information from VMs without installing an agent in the VMs was an open issue for a long time [77] [81] [78]. In [77], an agent-based architecture for VM software tracking is proposed. The detailed information about the VM execution was extracted in a transparent manner from the hypervisor and from within the VM. The resource consumption of the VM is extracted from the hypervisor level, but the VM application information is inferred by an agent installed inside the VM. Trihinas et al. in [78] propose a platform-independent and distributed monitoring architecture. Their monitoring solution focuses on the installation of a software agent for recovering all available metrics in the user cloud VMs. These agent-based techniques mostly use performance counters, and Linux tools like `iostat, vmstat, and top`, to gather statistics about the software applications running inside the VM [79] [80]. Not only do their methods need access to the VMs, but parsing and analyzing the output of these tools also bring extra overhead to the VMs. In [73], a vCPU state detection method, based on VMs tracing, was proposed. However, many companies, like Ravello systems [20], allows users to run unmodified nested VMs on existing cloud providers (like Google Cloud and Amazon AWS). Nested virtualization is commonly used for software testing and Continuous Integration (CI). Biancheri et al. in [108] extended this to multi-layer VM analysis (VM and Nested VM). Their traces consist in thousands of events, and looking into the events for each layer is a time consuming task. These agent-based techniques carry a very large overhead, as will be shown in the results section.

Calero et al. [81] proposed an agent-less monitoring architecture for cloud users in order to automatically see cloud resources. Their platform depends on OpenStack and does not scale well for many VMs. Virtual Machine Introspection (VMI) is another technique to analyse the running state of VMs without installing an agent inside the VM [83]. Although this technique is widely used for security threat analysis, it can also extract some monitoring metrics from the VM memory space. Several VMI techniques are proposed in [84] [85] for malware and security threat detection. Analyzing the memory space of a VM takes a significant time. Moreover, the overhead of VMs monitoring using VMI techniques is especially high if VMs

use a lot of memory. Furthermore, none of the VMI tools target VMs performance analysis in terms of resource usage and contention.

An agent-less adaptive SLA-based elasticity method was proposed in [82]. In their method, an end-to-end metric is used to scale up and down the number of CPUs. This method does not provide any reliable information about the disk or network. Another agent-less techniques was proposed in [111], [110], [114], and [109] to investigate the vCPU state. These methods, like the one proposed in this paper, are implemented based on hypervisor level tracing, and reveal the root cause of latencies in applications. In their approach, however, they do not provide information about blockings, once the VM is Idle.

Critical path analysis studies the dependencies between tasks in a project [115] and is widely used to identify sections to optimize. Similarly, in software engineering, critical path analysis is often used to assess resource bottlenecks, including for VM analysis. Once the critical path of processes inside VMs are identified, the IaaS provider is in a better position to understand the system and tune its performance, providing a better QoS. A precise approach for discovering a request dependency in a VM is used in vPath [86]. To use this method, the VMM should be modified to intercept each system call, and the VMs and host must be running Linux. Intercepting each system call adds further overhead as it needs to switch between guest and host modes back and forth. Moreover, the method does not provide any information about why the processes are waiting inside the VM.

The work which motivated the current research, is presented in [87]. In this paper, Giraldeau et al. present a technique to find the active path for the threads through different distributed machines. They can distinguish different states like wait for disk, network, timer, and tasks. Their approach could be applied to VMs but it would need to trace each VM individually. Then, they have to synchronize the traces and search through all the threads to discover the active path.

Early results about the new proposed method were presented in [116] [117]. We proposed techniques to detect the different states of vCPUs and VM processes. To the best of our knowledge, there is no pre-existing efficient technique to analyze the dependencies between the VM processes, the resources and the other processes. Through a graphical view of our VM process execution path, our technique displays the VM behaviour and provides a comprehensive insight to solve complex performance-related problems. Moreover, our technique incurs a lower overhead and is easier to deploy than existing methods, since it limits data collection to the host hypervisor level.

5.4 VM and Nested-VM Machine States

Modern Intel x86 (Intel-VT) CPU cores support the Virtual Machine eXtensions (VMX) instruction set to allow multiple operating systems (OSs) to share the CPU in a safe and efficient manner. Similarly, AMD supports the Secure Virtual Machine (SVM) instruction set. There are two VMX operation modes: the Virtual Machine Monitor is run in the **VMX root** mode. Guest software is executed in the **VMX non-root** mode. Consequently, there are two kinds of transition, from VMX root to VMX non-root, called **VM Entry**, and from VMX non-root to VMX root, called **VM Exit**. These transitions are controlled by an in-memory data structure called the Virtual-Machine Control Structure (VMCS). The VMM uses different VMCS for each vCPU. The VMM configures the VMCS structure using the `VMREAD`, `VMWRITE`, and `VMCLEAR` instructions [17]. The performance overhead associated to a VM entry/exit is the time spent to read/write from/into the VMCS structure, to complete context switching, and the added cache faults due to running the hypervisor code.

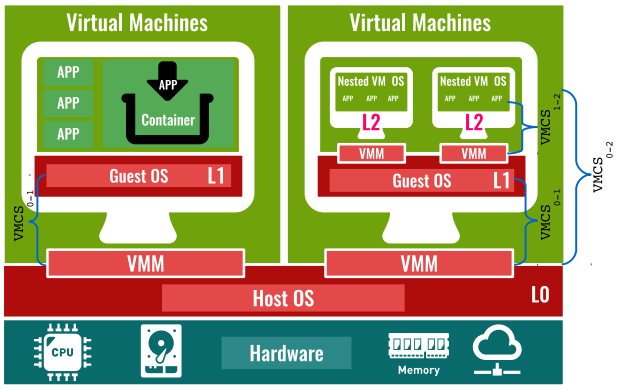


Figure 5.1 Two-Level VMs (Nested VM) Architecture for VMX

Figure 5.1 shows the architecture of one-level VMs and two-level VMs (nested) respectively. In one-level VMs, one pointer to the VMCS structure is used for each vCPU of each VM. However, for two-level VMs, there are three VMCS structures for each vCPU. The first VMCS is created to contain the information between the host and the first-level VM, and another VMCS is created for storing the information between the VM and the nested VM. Finally, the last VMCS is constructed to store the information between the host and the Nested VM.

Indeed, in a two-level architecture, executing any privileged instruction in the second level (L2) traps to the host hypervisor (L0). The VM hypervisor (L1) has the illusion of running its code and the nested VM code (L2) directly on a physical pCPU. In most cases, the host hypervisor (L0) updates the VM hypervisor (L1) VMCS by a transition from L0 to L1. After

updating the VM hypervisor, the host hypervisor resumes the nested VM directly. In this project, we take into account the hierarchical architecture of VMs and detect the different states (Wait_For and Running) for the code in the VM and in the Nested VM. In addition, we construct the execution graph of the nested VMs and its interaction with the VMs and Host.

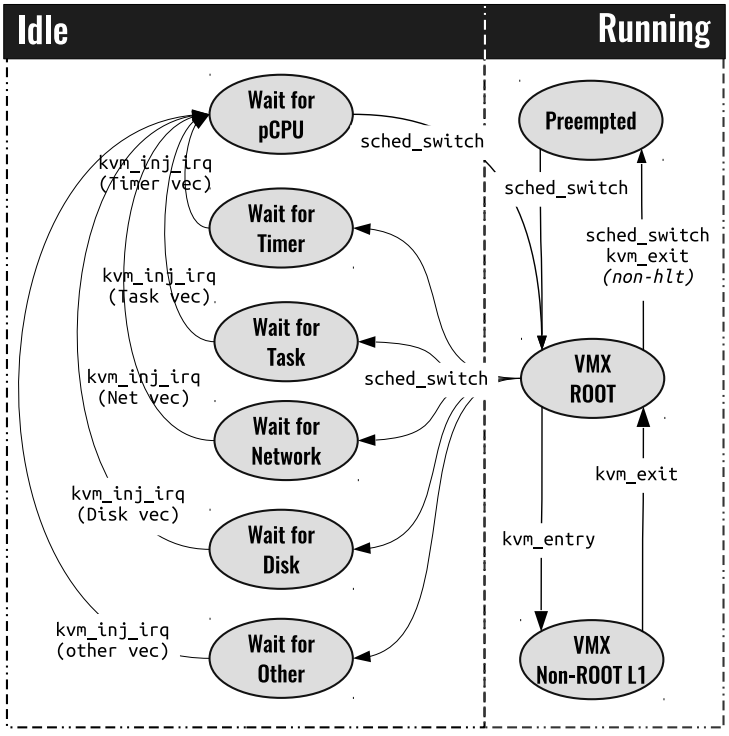


Figure 5.2 One-Level Virtual Machine Process State Transition

5.4.1 Virtual Interrupt

Intel-VT supports virtual interrupts (`virq`) by setting a control bit in the VMCS. When an interrupt is injected into the vCPU, the vCPU uses a different Interrupt Descriptor Table (IDT) than the one used in the host. Receiving an interrupt causes a VM exit, and then the VMM injects a `virq` into the VM by emulating the *LAPIC* register. Once the VM resumed, the pending interrupt is executed by looking up the VM IDT. After handling the interrupt, the guest writes in the emulated EOI register, which causes an exit to the VMM root mode.

5.4.2 Virtual Process States

Figure 5.2 shows the different states of a vProcess and the conditions to transition to those states. The running process states alternates between the VMX root and VMX non-root

states. Processes waiting for a resource are in the Preempted, and wait_for_X states. The Preempted state is when the pCPU is taken from the vCPU and is given to another process in the host (including another vCPU in the same or another VM). Other wait_ for_X states occur when the vCPU is being scheduled out voluntarily by sending the `hlt` signal from the guest OS. In these states, the vProcesses are waiting for a wake-up signal. A vProcess can be woken up for the following reasons. **First**, a process inside the VM sets a timer and the timeout occurred (Timer Interrupt). **Second**, a process inside the VM is awakened by another process (IPI Interrupt). **Third**, a process inside the VM is woken up by a remote task over a socket (Network Interrupt). **Fourth**, a process inside the VM is waiting for the disk (Disk Interrupt). The interrupt later injected into the VM reveals the reason for the idle state. **Fifth**, a process inside the VM could also wait for another device (with different virq number), in which case we label it as wait_for_other. Figure 5.3 depicts the states of a process inside a n-level nested VM. As shown, the process could be executing in any level, and could be preempted in any level of virtualization. Therefore, a process in any level could wait for a resource.

5.5 VM Analysis Algorithms

This section presents three algorithms to analyze VMs. **First:** the Any-Level vCPU States Detection Algorithm (ASD) to compute the different states of vCPUs for arbitrary nesting depths. ASD is based on scrutinizing incoming events from the Host Hypervisor to identify nested VMs and corresponding vCPU states. **Second:** the Guest (Any Level) Thread-state Analysis (GTA), to reveal hidden information about threads and processes inside the VMs. GTA, like ASD, uncovers the wait and running states for the processes in any level of nesting. **Third:** the Host-based Execution-graph Construction (HEC) Algorithm computes the execution path of an arbitrary process inside VMs and through network messages (at any level).

Our algorithms receive as input events collected by tracing the host hypervisor. Tracing consists in collecting event data when the tracer encounters an enabled tracepoint at runtime. Tracing is useful to monitor a program execution without stopping it. An event carries a timestamp and a payload. The timestamp tells when the tracepoint was encountered, and the payload contains extra information about the event. For example, the payload of a context switch event contains the Process name, ID, and priority of the two switched tasks.

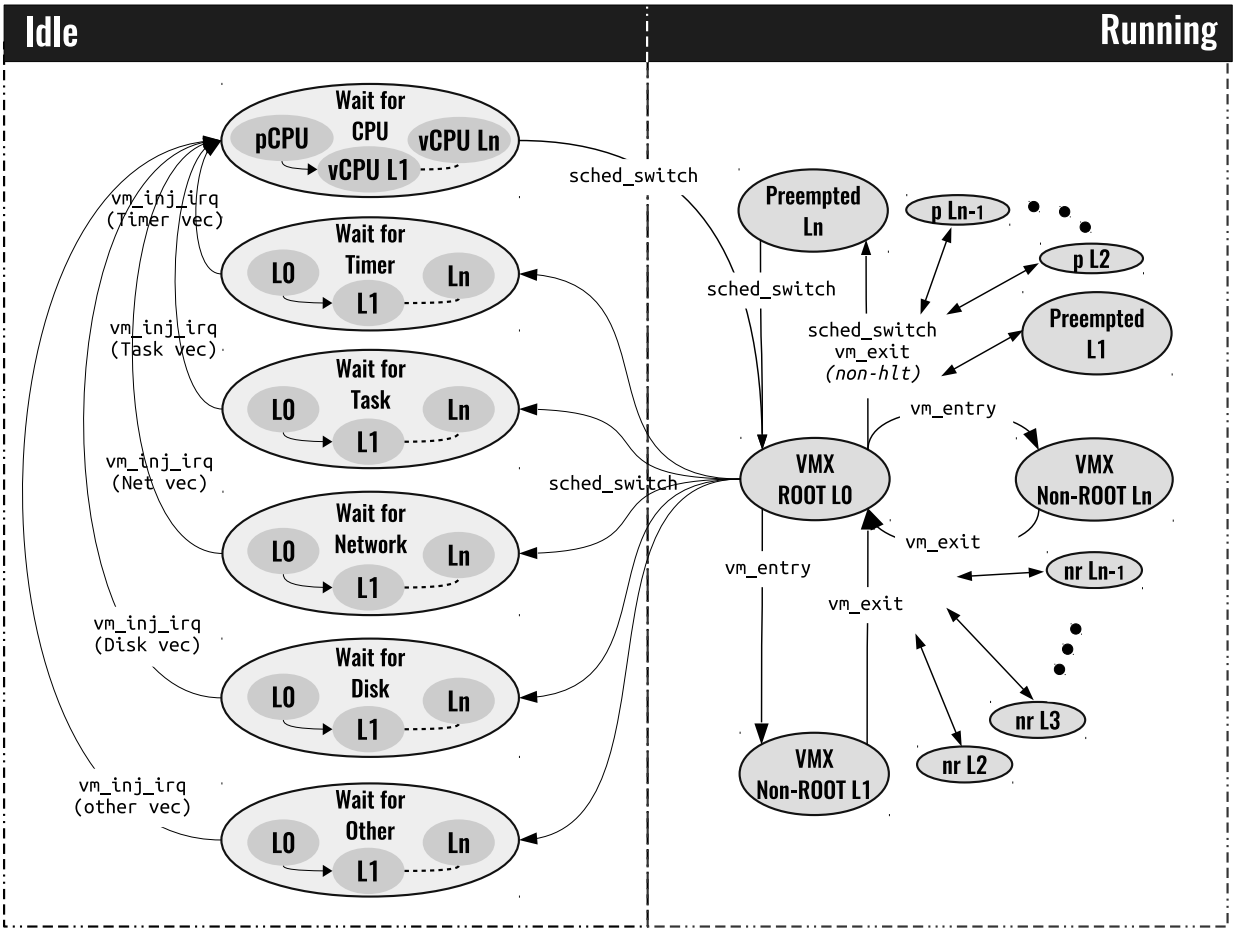


Figure 5.3 n-Levels Nested Virtual Machine Process State Transition

5.5.1 Any-Level vCPU States Detection Algorithm (ASD)

In this subsection, we propose an algorithm to uncover vCPU states (all running and waiting states) for arbitrary depths of nesting for the vCPU threads of VMs. Before introducing the Any-Level vCPU States Detection Algorithm (ASD) pseudocode, we illustrate it with a simple example. The simplest algorithm is called the Entry-Exit Algorithm (2EA), which uses the `sched_in, sched_out, vm_enter, and vm_exit` events. The `sched_in` represents when a vCPU gets onto a pCPU. In reverse, the $\texttt{sched\_out}_1$ schedules out the VM1 vCPU from the pCPU. The $\texttt{vm\_enter}_1$ and $\texttt{vm\_exit}_1$ mark the point at which a transition is made between the VM1 and the Hypervisor. Figure 5.4-a presents an example of the 2EA algorithm, using a sequence of events shown in Table 5.1. At time 10, the `vm_exit` event causes a transition from vmx non-root to vmx root and then yield the pCPU by receiving sched_out at time 12. The next event is `sched_in`, which shows that the vCPU1 is being scheduled in

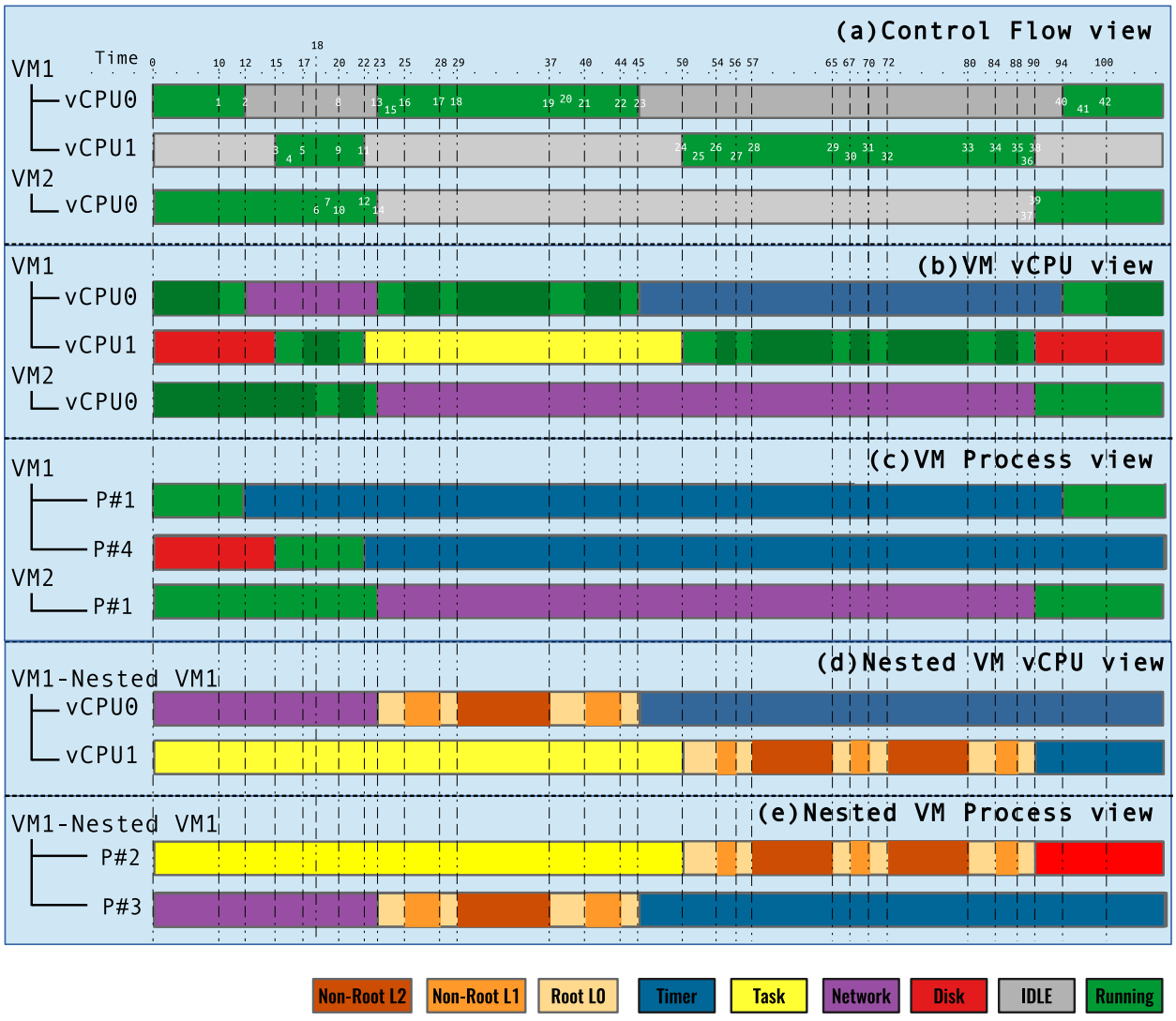


Figure 5.4 vCPU and VM Process Views using different algorithms

and the state changes to vmx root. When receiving the `vm_entry` event, the state changes to vmx non-root. This algorithm is simplistic and cannot detect other virtual levels and states. For example, there is no information about the reason for waiting, to reveal the vCPU and nested VM information.

Figure 5.4-b and Figure 5.4-d present a vCPU view using the ASD algorithm. These views show the exact level of code execution and wait states. As shown with the 2EA algorithm, the VM1-vCPU1 was Idle between time 0 and 15, and the reason for waiting was not detected. The key idea is that the event indicating the cause for the Idle state is unknown a priori. The ASD algorithm uses the $\texttt{inj\_virq}_1$ to reveal the Idle state. When receiving the $\texttt{sched\_in}_1$

Table 5.1 Events and their payload based on Host kernel tracing ($vec_0 = Disk, vec_1 = Task, vec_2 = Net, vec_3 = Timer$)

#	TimeStamp & Events	#	TimeStamp & Events
1	10 vm_exit$_1$(*reason=12*)	22	44 vm_exit$_1$(*reason=12*)
2	12 sched_out$_1$(*vCPU0*)	23	45 sched_out$_1$(*vCPU0*)
3	15 sched_in$_1$(*vCPU1*)	24	50 sched_in$_1$(*vCPU1*)
4	16 inj_virq$_1$(vec_0)	25	52 inj_virq$_1$(vec_1)
5	17 vm_enter$_1$(*CR3 P#4*)	26	54 vm_enter$_1$(*CR3 P#5*)
6	18 vm_exit$_2$(*reason=30*)	27	56 vm_exit$_1$(*reason=24*)
7	19 net_sock_in$_2$(*pkt#1*)	28	57 vm_enter$_1$(*CR3 P#2*)
8	19 net_sock_out$_2$(*pkt#1*)	29	65 vm_exit$_1$(*reason=24*)
9	20 vm_exit$_1$(*reason=12*)	30	67 vm_enter$_1$(*CR3 P#5*)
10	20 vm_enter$_2$(*CR3 P#1*)	31	70 vm_exit$_1$(*reason=12*)
11	22 sched_out$_1$(*vCPU1*)	32	72 vm_enter$_1$(*CR3 P#2*)
12	22 vm_exit$_2$(*reason=12*)	33	80 vm_exit$_1$(*reason=12*)
13	23 sched_in$_1$(*vCPU0*)	34	84 vm_enter$_1$(*CR3 P#5*)
14	23 sched_out$_2$(*vCPU0*)	35	88 vm_exit$_1$(*reason=12*)
15	24 inj_virq$_1$(vec_2)	36	89 net_sock_in$_1$(*pkt#1*)
16	25 vm_enter$_1$(*CR3 P#5*)	37	89 net_sock_out$_2$(*pkt#1*)
17	28 vm_exit$_1$(*reason=24*)	38	90 sched_out$_1$(*vCPU1*)
18	29 vm_enter$_1$(*CR3 P#3*)	39	90 sched_in$_2$(*vCPU0*)
19	37 vm_exit$_1$(*reason=12*)	40	94 sched_in$_1$(*vCPU0*)
20	38 sched_ttwu$_1$(*vCPU1*)	41	97 inj_virq$_1$(vec_3)
21	40 vm_enter$_1$(*CR3 P#5*)	42	100 vm_enter$_1$(*CR3 P#1*)

event at time 15, the VM1-vCPU1 goes to the root L0 state. The `inj_virq`$_1$ event exposes the reason for waiting by interpreting the *vec0* which is waiting for disk. The `vm_enter`$_1$ event at time 17 changes the state to non-root L1. Upon receiving `vm_exit`$_1$ at time 20, with exit reason 12, the state goes to root L0. The next event is `sched_out`$_1$, which shows that the VM1-vCPU1 is being scheduled out, and the state changes to unknown. This unknown state will change to wait for task at time 52, when receiving `inj_virq`$_1$.

Using the ASD algorithm, the exact level of code execution and the reasons for wait state can be detected. 5.4-d presents an example of all the states for the vCPUs of a nested VM. When receiving the `sched_in`$_1$ event at time 23, the VM1-vCPU0 status goes to L0. The `inj_virq`$_1$ event exposes that the VM1-vCPU0 of VM was waiting to receive a network packet. The `vm_entry`$_1$ event at time 25, changes the status to L1 and the CR3 value of the guest is stored as the expected hypervisor CR3. The CR3 register points to the page directory of a process, in any level of virtualization, and can be used as a unique identifier for a process. The next event, `vm_exit`$_1$, modifies the status of VM1-vCPU0 to L0 and, since the exit reason is vm resume (exit reason 24), it pops the CR3 value from the expected hypervisor stack. In this case, the expected hypervisor CR3 is marked as the CR3 of the hypervisor in L1. Upon receiving the 12th event, the vCPU status goes to L2 and the wait reason for the nested VM changes to wait for network. The ASD algorithm shows that the received packet belongs to the nested VM, inside the VM. As mentioned, executing any privileged instruction in any level of nested VMs causes an exit to L0. Therefore, the `vm_exit`$_1$ modifies the vCPU state

to L0. The algorithm uses the marked CR3 to distinguish between L1 and L2. The same approach is used to reveal the VM2-vCPU and VM2-process states.

The pseudocode for the ASD algorithm is depicted in Algorithm 5. The ASA algorithm receives a sequence of events as input and updates the vCPU state. In this algorithm, we uses an array of List of hypervisors for level n. The candidates[m] (m is the level of nesting available) variable is an array of stacks which holds the hypervisor candidate for level n. We also use different HashMaps to keep some information about VMs and Nested VMs.

In case the event is `wake_up`, the state of the vCPU is modified to the Wait for pCPU state (w4pCPU) (Line 11). The most important event, `sched_switch`, shows when a vCPU is running on a pCPU. When a vCPU is scheduled in, it goes to the L0 state (Line 17) to load the previous state of the VM from the VMCS. By contrast, when a vCPU is scheduled out, it checks the last exit reason to go either to the wait for unknown state, with *hlt* reason (Line 22), or to the *preempted* state (Line 24). The `getLastExit(vCPU)` function returns the last exit for the existing vCPU. If the last_exit is *hlt*, which means that the VM runs the idle thread, the state will be modified as wait for unknown (w4Unkown). For the other exit reason cases, it will be changed to the preempted state.

When receiving a `vm_entry` event, the state of the vCPU is adjusted to VMX non-root state. First, it uses the `getCandidateLevel()` function to find out the level of entry. The `getCandidateLevel()` function uses CR3, lCR3 (last hypervisor CR3), and a HashMap variable (levels) to find out the code execution level for CR3. It compares CR3 with the available hypervisor lists in all levels to find out if the CR3 belongs to an hypervisor. In case CR3 is not found in the hypervisor list, it checks lCR3 to find out the last level of code execution (Line 28). The last exit reason is compared if it is VMRESUME or VMLAUNCH (Line 30). If the condition is true, it means that the VM in level n is running another VM. As a result, the last CR3 that was pushed onto the hypervisor candidate list will be popped (Line 32), and will be added to the hypervisors list in level n (Line 34).

If the condition is false, the CR3 will be pushed to the candidate list for level n (Line 40). After calculating the level at the end of this event, the state of the vCPU will be updated to L_n (Line 43). Receiving any `vm_exit` changes the status to L0 (Line 46). In case the event is `inj_virq`, the vec field in this event is compared with the Task, Timer, Disk, and Network interrupt number. The unknown state is updated based on the *vec* number.

We implemented the proposed algorithm in Trace Compass [2] as a new graphical view for vCPU threads inside the host, and vCPUs inside the VM. The vCPU state is stored in a disk-based database named State History Tree (SHT). In the next section, we present the results of some experiments using the ASD algorithm.

Algorithm 5: Any-Level vCPU States Detection Algorithm (ASD)

Input : Trace $\mathcal{T}$
Output: State of $vCPUs$

```
1  ---------- Initialization ----------
2  vCPUs ⟵ {inital tid}
3  HashMap levels, lastCR3, lExit, timing, vCPU, states;
4  List hypervisorsList[m] ;
5  Stack candidates[m];
6  ---------- Main Procedure ----------
7  for all event e ∈ T do
8      if e.type is wake_up then
9          j = getVMvCPU(e.tid);
10         if tid == vCPU_j^tid then
11             vCPU_j^State = w4pCPU ;
12             updateVCPUTiming(e.tid,w4pCPU,e.ts)
13     else if e.type is sched_switch then
14         k = getVMvCPU(e.prev_tid);
15         j = getVMvCPU(e.next_tid);
16         if next_tid == vCPU_j^tid then
17             vCPU_j^State = L0 ;
18             updateVCPUTiming(e.next_tid,Root_L0,e.ts)
19         if prev_tid == vCPU_k^tid then
20             last_exit = getLastExit(vCPU_k^tid) ;
21             if last_exit == hlt then
22                 vCPU_k^State = w4Unkown ;
23             else
24                 vCPU_k^State = Preempted_L0
25     else if  e.type is vm_entry then
26         last_exit = getLastExit(vCPU_j^tid) ;
27         lCR3 = getLastCR3(vCPU_j^tid) ;
28         n = getCandidateLevel(CR3,lCR3) ;
29         hypervisors = getHypervisorsList(n);
30         if last_exit == VMRESUME or VMLAUNCH then
31             cHyperviors = getCandidateStack(n);
32             hCR3 = cHyperviors.pop();
33             if !hypervisors.contains(hCR3) then
34                 hypervisors.add(hCR3) ;
35                 putHypervisorList(n,hypervisors) ;
36                 levels.put(hCR3,n);
37                 levels.put(CR3,n+1) ;
38             n ++;
39         if !hypervisors.contains(CR3) then
40             putCandidateStack(n,CR3) ;
41         waitState = queryWait4(vCPU_k^tid);
42         updateWaitState(vCPU_k^tid,waitState);
43         vCPU_k^State = Ln ;
44     else if  e.type is vm_exit then
45         putLastExit(vCPU_j^tid, exit_reason) ;
46         vCPU_j^State = L0 ;
47     else if  e.type is inj_virq then
48         j = getVMvCPU(e.tid);
49         if e.tid == vCPU_j^tid then
50             if e.vec == Task Interrupt then
51                 updateWait4(vCPU_j^tid,w4Task);
52             else if e.vec == Timer Interrupt then
53                 updateWait4(vCPU_j^tid,w4Timer);
54             else if e.vec == Disk Interrupt then
55                 updateWait4(vCPU_j^tid,w4Disk);
56             else if e.vec == Network Interrupt then
57                 updateWait4(vCPU_j^tid,w4Network);
```

Algorithm 6: Utility Functions for ASD and GTA algorithm

```
1  Function getCandidateLevel(currentCR3, lastCR3):
2      int n;
3      if lastCR3 == unknown then
4          n = 0;
5      else
6          n = levels.get(currentCR3);
7      return n;
8  Function putCandidateStack(n, CR3):
9      candidates[n].push(CR3);
10 Function getCandidateStack(n, CR3):
11     return candidates[n].pop();
12 Function getHypervisorsList(n):
13     return hypervisorsList[n].pop();
14 Function getLastCR3(vCPU_j):
15     return lastCR3.get(vCPU_j);
16 Function getLastExit(vCPU_j):
17     return lastExit.get(vCPU_j);
18 Function getVMvCPU(tid):
19     return vCPU.get(tid);
20 Function updateVCPUTiming(tid, state, ts):
21     k = getVMvCPU(e.tid);
22     vCPUTiming = timing.get(vCPU_k);
23     vCPUTiming.push(state,ts) ;
24     timing.get(vCPUTiming) ;
25 Function updateWaitState(vCPU_k, state):
26     vCPUTiming = timing.get(vCPU_k);
27     while (!vCPUTiming.empty()) do
28         [state,ts] = vCPUTiming.pop();
29         vCPU_k^State(ts) = state ;
30 Function queryWait4(vCPU_k):
31     return states.get(vCPU_k) ;
32 Function updateWait4(vCPU_k, waitState):
33     states.put(vCPU_k, waitState) ;
34 Function updateProcessWait(vCPU_k,CR3,n):
35     processIdleState = states.get(vCPU_k) ;
36     Process_n^CR3,SP = processIdleState ;
```

5.5.2 Guest (Any Level) Thread-state Analysis (GTA)

As mentioned in the previous section, for each transition between the root mode and non-root mode, the processor state is saved in the VMCS fields. The guest state is stored in the guest-state area in each `vm_exit` and is loaded from the guest-state area at each `vm_entry`. Also, the host state is retrieved at each `vm_exit`. The instruction pointer (IP), stack pointer (SP) and control registers (CR) are some of the registers that are modified during each transition in the VMCS guest-state area.

The process identifier (PID) and process name of each thread inside the guest are not directly accessible from host tracing. The only information which can easily be uncovered by host tracing about the threads inside the VMs is written in CR3 and SP. CR3 and SP can uniquely identify the process and thread, respectively. Indeed, CR3 points to the page directory of a process in any level of virtualization. All threads within a given process use the same page directory. Therefore, switching between two threads within the same process does not change the CR3 value. SP points to the stack of the thread inside the VM. As a result, by retrieving these two identifiers, we can find out which thread of which process is executing on a vCPU. To have more information about the threads inside the VMs or Nested-VMs, we could map CR3 and SP to the thread identifier (TID), PID and process name. This is not strictly necessary, since CR3 and SP are unique identifiers for the threads and processes, but it would be more convenient and human readable if we map the process information inside the guest to the information we get from the `vm_entry` trace point.

Figure 5.4-c and 5.4-e show the VM process and Nested VM process views using the Guest (Any Level) Thread-state Analysis (GTA). The algorithm uses the CR3 and SP fields in `vm_entry` events to distinguish between different processes and threads. When receiving a `vm_entry` event at time 17, the state of process with CR3#4 changes to running as vmx non-root, and with `vm_exit` at time 20, the state goes to vmx root. The GTA algorithm queries the data stored for vCPU views to figure out the level of running processes. For example, at time 29, it checks the level of code execution and finds out that process CR3#3 is a process in the nested VM.

The GTA algorithm is illustrated in Algorithm 7. To simplify presentation, error handling is not shown and the data structures are shown for one VM per host. When the event is `vm_entry`, the stack pointer and CR3 of that thread are gathered from the VMCS guest state, and the process information of the VM is updated. If the mapping information from the VM is available, the CR3 and SP values are converted to the name of the process and threads (Line 3). With `vm_entry` and `vm_exit` events, it queries the state of the vCPU to update the state of the process and thread running on that vCPU (Line 9,14).

Algorithm 7: Guest (Any Level) Thread-state Analysis (GTA)

Input : Trace $\mathcal{T}$
Output: State of *Processes*

```
1  ____________________ Initialization ____________________
2  if Mapping == TRUE then
3  |  Map SP and CR3 with process memory map of VM ;
4  ____________________ Main Procedure ____________________
5  for all event e ∈ T do
6  |  j = getVMvCPU(e.tid);
7  |  if e.type is vm_entry then
8  |  |  n = getProcessLevel(CR3,lCR3) ;
9  |  |  status = queryStatus(vCPU_j) ;
10 |  |  Process_n^{CR3,SP} = status ;
11 |  else if e.type is vm_exit then
12 |  |  n = getProcessLevel(CR3,lCR3) ;
13 |  |  updateProcessWait(CR3,n);
14 |  |  status = queryStatus(vCPU_j) ;
15 |  |  Process_n^{CR3,SP} = status ;
```

5.5.3 Host-based Execution-graph Construction (HEC) Algorithm

We compute the critical path of an execution using the Host-based Execution-graph Construction (HEC) Algorithm. The input to the algorithm is the trace events and the output is the processes dependency graph. In addition, the algorithm outputs the critical path of the process executions, characterized by process id, starting timestamp, ending time-stamp, and segments status. The status could be running (at a certain level of code execution), preempted (in different levels), and wait for OS (block device, network or timer).

The HEC algorithm is presented in Algorithm 8. Here again, error handling is not shown and the data structures are shown for one VM per host. However, the algorithm can detect and find processes dependencies for any number of VMs per host. The *execution graph* data structure is a two-dimensional doubly linked list, where the horizontal links are the start and end of task states, and the vertical edges represent wake-up signals between processes. The algorithm first initializes the state of running processes by creating a vertex for the process with a time-stamp for the beginning of the trace (Line 7). Then, the algorithm iterates over the events and generates new vertices according to their types. The `sched_switch` event adds one more new edges for previous processes, as running in the host hypervisor level, to the graph (Line 13). Then, the algorithm checks if the last exit for the previous process is `hlt` or not (Line 16). In the case where `hlt` was the last exit state, the process was blocked for reasons other than preemption. If the process was preempted, the `vm_entry` event adds

one new edge for the process as preempted to the graph (Line 22). Otherwise, a new edge is added for the process as running in the host hypervisor level. When receiving a `vm_exit` event, a new horizontal link is added to the previous process based on the code execution level. The `sched_ttwu` event sets the waker and wakee processes to be used later when processing a `inj_virq` event (Line 34). The `inj_virq` event adds vertical and horizontal edges based on the IRQ number. If the IRQ number matches IPI, a vertical edge would be added from waker to wakee (Line 42). In addition, it adds two new horizontal edges to the graph, one for the waker that is blocked (Line 40), and the other for the wakee that is running (Line 41). When the IRQ number is equal to Timer interrupt, Disk interrupt, or Network interrupt, an horizontal edge is added as Timer (Line 44), and Disk (Line 46), respectively. When receiving `inet_sock_local_in` and `inet_sock_local_out` events, the algorithm finds the matched packet and adds a vertical link between the sender and receiver of the packet(Line 47-52). The complexity of the HEC algorithm is *O(n)*, where n is the number of events in the trace.

To find the active path of execution for a process inside the VM, all blocking edges are replaced by their corresponding processes. We use the Active Path Computation algorithm proposed in [87] to compute the *active path of execution*. The Active Path of Execution (APE) algorithm starts with the start vertex and traverses the graph recursively .

To get a better understanding of how the HEC algorithm works, a simple example is provided. Figure 5.5-a presents the execution graph of processes inside a VM, based on events from Table 5.1. It shows the horizontal links between different states of the processes. In order to find the active path of each process, we use the APE algorithm. Figure 5.5-a depicts the full dependency graph for all the processes in different VMs. Figure 5.5-b shows the active path for process #1. As shown, the process waits for the timer during interval [12-94] and then it executes again. The active path for process #4 is presented in Figure 5.5-c. As shown, process #4 waits for disk and then runs from time 15 to 22. Then, it waits for a timer to fire. The first interesting active path is shown in Figure 5.5-d for process #2 where it waits for process #3 during interval [0-50]. As a result, the active path of process #2 includes process #3. Actually, it is shown that process #3 was blocked for a network packet. When receiving the packet, it wakes up process #2. The active path for process #3 is presented in 5.5-e. Process #3 waits for a network packet and executes for time interval [23-45]. Then, it waits for a timer to be fired. Finally, the second interesting active path is depicted in Figure 5.5-f where VM2 process #1 waits for Nested-VM1↪process #2 and Nested-VM1↪ process #3.

Algorithm 8: Host-based Execution-graph Construction (HEC) Algorithm

Input : Trace $\mathcal{T}$
Output: Execution Graph $\mathcal{G}$

```
1  ──────────────────────────────────────── Initialization ────────────────────────────────────────
2  PROCESS ⟵ {inital CR3}
3  for all process p ∈ PROCESS do
4      if p.state is running then
5          vcpu = get_vcpu(p.tid);
6          set_vcpu(vcpu, p.cr3);
7      Create initial vertex of process p with timestamp T.begin
8  hec:sock:out ──────────────────────────────────── Main Procedure ────────────────────────────────
9  for all event e ∈ T do
10     if (e.type is sched_switch) then
11         j = getVMvCPU(e.prev_tid);
12         pCR3 = getLastCR3(vCPU_j);
13         LINK_HORIZONTAL(pCR3, e.ts, L0);
14         k = getVMvCPU(e.next_tid);
15         lExit = getLastExit(vCPU_k);
16         if lExit != hlt then
17             updateCR3Status(pCR3, preempted);
18     else if (e.type is vm_entry) then
19         j = getVMvCPU(e.tid);
20         pCR3 = getLastCR3(vCPU_j);
21         if (queryCR3Status(e.cr3) == preempted) then
22             LINK_HORIZONTAL(e.cr3, e.ts, PREEMPTED);
23         else
24             LINK_HORIZONTAL(e.cr3, e.ts, L0);
25     else if (e.type is vm_exit) then
26         j = getVMvCPU(e.tid);
27         pCR3 = getLastCR3(vCPU_j);
28         levelState = queryLastLevel(vCPU_j);
29         LINK_HORIZONTAL(pCR3, e.ts, levelState);
30     else if (e.type is sched_ttwu) then
31         j = getVMvCPU(e.wakee_tid);
32         k = getVMvCPU(e.waker_tid);
33         wakerCR3= getLastCR3(vCPU_k);
34         updateWaker(vCPU_j, wakerCR3);
35     else if (e.type is inj_virq) then
36         j = getVMvCPU(e.tid);
37         pCR3 = getVMvCPU(vCPU_j);
38         if (e.irq is IPI) then
39             wakerCR3 = queryWaker(vCPU_j);
40             V_1 ⟵ LINK_HORIZONTAL (wakerCR3, e.ts, BLOCKED);
41             V_2 ⟵ LINK_HORIZONTAL (pCR3, e.ts, L0);
42             LINK_VERTICAL(V_1, V_2, TASK);
43         else if (e.irq is timer) then
44             LINK_HORIZONTAL (pCR3, e.ts, TIMER);
45         else if (e.irq is disk) then
46             LINK_HORIZONTAL (pCR3, e.ts, DISK);
47     else if (e.type is inet_sock_local_in) then
48         tx ⟵ LINK_HORIZONTAL (pCR3, e.ts, L0);
49     else if (e.type is inet_sock_local_out) then
50         if (packet match found) then
51             rx ⟵ LINK_HORIZONTAL (pCR3, e.ts, L0);
52             LINK_VERTICAL(tx, rx, NETWORK);
```

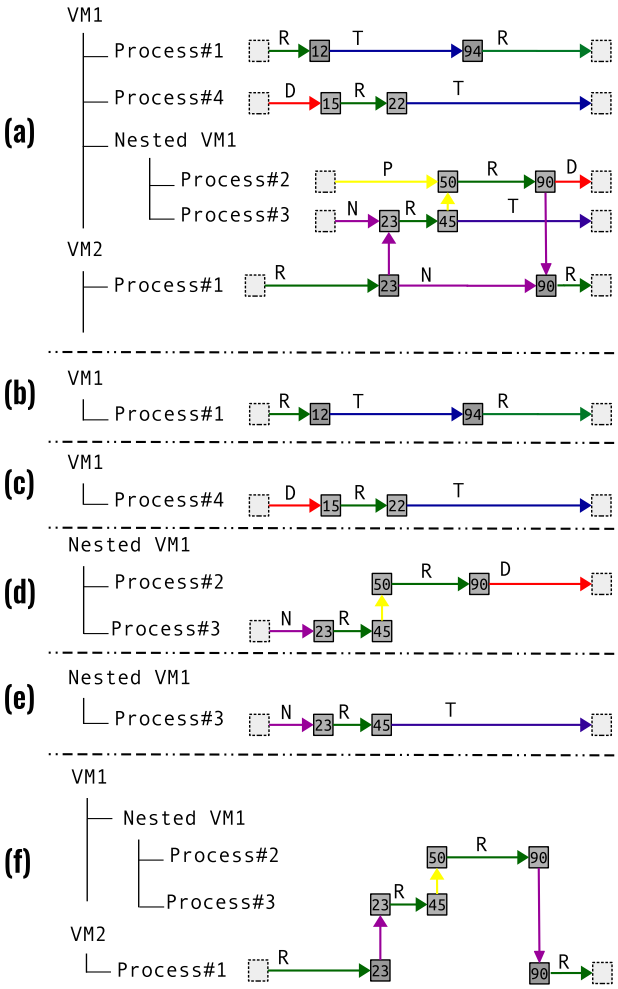


Figure 5.5 Example of Execution Graph based on events from Table 5.1

5.6 Use cases

This section covers a variety of known problems that our technique is able to detect. First, we use the ASD and GTA algorithms to detect issues related to waiting for different resources. To illustrate a wait for disk, we conducted an experiment on *Hadoop TeraSort*, since it is an excellent example of real world distributed I/O intensive job. To analyze a wait for network, we present a RPC client and server, a very common scenario because of micro-services. To investigate wait for CPU and memory, the *sysbench* benchmarking tool, along with a c-based application, is used in order to put pressure on the CPU and memory, with sysbench, and observe the effect on the application.

The ASD and GTA algorithms are also used to analyse the sensitivity of different VMs to resources availability. We conducted our sensitivity analysis on well-known Linux servives,

using the widely used MySQL and Apache servers.

Then, the HEC algorithm was evaluated to reproduce and examine a known problem with the popular APT Linux packaging tool. Thereafter, we evaluated our VM tracing technique for detecting issues in multi-thread applications, to show DNS network latencies in IMS networks, and to analyse the boot-up of different OSs. These usecases are representative of issues that we have encountered in industry. Finally, we compare our technique with other existing methods in terms of overhead and accuracy.

5.6.1 Analysis Architecture

Our approach is independent of the Operating System (OS) and could work on different architectures using Intel or AMD processors. Due to good nesting support, we have chosen, for our use cases, KVM under the control of OpenStack, which is the most commonly used hypervisor for Openstack [112]. Our experiments with nested VMs are limited to one level of nesting because of KVM. Qemu is used for the userspace part of hypervisor. We also use the same architecture for nested VMs and the VM hypervisors. The events are collected by the tracer (LTTng) at the host hypervisor level, and then the events are sent to the trace analyzer (Trace Compass). Our experimental setup is described in Table 5.2. Qemu version 2.5 and the KVM module of Linux kernel 4.2.0-27 were used.

Table 5.2 Experimental Environment of Host, Guest, and NestedVM

	Host Environment	**Guest Environment**	**Nested VM Environment**
CPU	Intel(R) i7-4790 CPU @ 3.60GHz	Two vCPUs	Two vCPUs
Memory	Kingston DDR3-1600 MHz, 32GB	3 GB	1 GB
OS	Ubuntu 15.10 (Kernel 4.2.0-27)	Kernel 4.2.0-27	Kernel 4.2.0-27
Qemu	v2.5	v2.5	-
LTTng	v2.8	v2.8	v2.8

The analysis is performed following these steps:

1. Start tracing on the host;
2. Run the VMs of interest;
3. Stop tracing;
4. Run the analyses;

5. Display the result in the interactive viewer;

Among the available Linux tracers, we choose a lightweight tracing tool called the Linux Tracing Toolkit Next Generation (LTTng) [1] due to its low overhead kernel and userspace tracing facilities. Furthermore, in Linux, the KVM module is instrumented with static tracepoints and LTTng has appropriate kernel modules to collect them. Therefore, LTTng is particularly suitable for our experiment, since it collects Linux kernel and KVM module events with a low impact on VMs. We also added our own tracepoint (vcpu_enter_guest) to retrieve the CR3 and SP values from the VMCS structure, on each VMX transition, using kprobe. After the relevant events are generated and collected by LTTng, we study those with the trace analyzer, as elaborated in the next subsection. The events required for the analysis, and the instrumentation method, along with their name in LTTng, are shown in Table 5.3.

We implemented our event analyzers as separate modules in Trace Compass [2]. Trace Compass is an Open-source software for analyzing traces and logs. It provides an extensible framework to extract metrics, and to build views and graphs.

5.6.2 Use case 1: Wait for Resources

Predicting VM workloads is an open challenge for both cloud providers and cloud users. Statically partitioning the physical resources between VMs, based on peak VM demands, usually leads to a waste of resources, increasing capital expenses. Alternatively, to achieve high resource utilisation, the cloud provider may over-commit the resources, sharing them among too many VMs. In that case, the VMs might contend on a shared resource, causing undue latency. In order to deal with this issue, a resource cap is applied to each VM. Most Cloud providers, like Amazon, limit the resource usage for each VM in order to prevent important performance reductions due to sharing resources. For example, the IO size is capped

Table 5.3 Events required for analysis

Category	Event	LTTng Event	Method
scheduler	sched_switch	sched_switch	tracepoint
scheduler	wake_ttwu	sched_ttwup	tracepoint
hypervisor	vm_exit	kvm_exit	tracepoint
hypervisor	vm_inj_virq	kvm_inj_virq	tracepoint
hypervisor	vm_entry	kvm_entry	tracepoint
hypervisor		vcpu_enter_guest	kprobe
network	inet_sock_in	inet_sock_local_in	netfilter
network	inet_sock_out	inet_sock_local_out	netfilter

at 256 KiB for SSD volumes and 1024 KiB for HDD volumes in Amazon EC2 instances, with a certain number of I/O operations per second allowed [118]. Sometimes, the cloud provider sets the VM resource cap too low, which leads the cloud user to experience Service Level Objective (SLO) violations (local impact on applications). If the resource cap is too high, however, the cloud provider must pay for the wasted resources. To avoid these problems, elastic resource capping is introduced, but brings more challenges like "How much/When the resource cap should be increased/decreased?".

In the next subsections, we show how our technique could detect resource overcommitment, and issues related to setting the VM resource cap too high/low.

5.6.2.1 Wait for Disk Issue

Recent processors use many techniques like pipelining and multi-threading to efficiently execute application code. However, the block I/O operations may slow down the execution of applications while the CPU waits for the data. It also frees the CPU for other processes to execute. Discovering precisely why and when a process is waiting for disk is a big challenge, especially when dealing with several layers of virtualization. In the following experiments we address the orthogonal problems of: Is there any I/O bottleneck in the virtualization environment? What impacts I/O performance? Where is the root cause of an I/O bottleneck?

The first experiment is conducted on a large scale distributed computing application across a clustered system where each VM had 3 vCPUs with 3GB of memory. The data was one million numbers (50GB) to be sorted by *Hadoop TeraSort* (mix of CPU-intensive and I/O intensive job). In our Hadoop configuration, one VM is designated as master and 2 other VMs are configured as slaves. The Hadoop version was 2.8.1 and the java version was 8. The Yarn framework is used for job scheduling and resource management. After distributing the job to different instances, we realized that the total execution time for sorting the numbers is larger than expected. We reproduced the problem by submitting the same data set to *Hadoop TeraSort* while tracing the host with LTTng. In our investigation with the ASD algorithm, we found that the VM-Slave 2 finished its jobs faster than VM-Slave 1. By applying the GTA algorithm, we found three significant processes in each VM and, since the processes of VM-Slave 1 waits more for disk, they responded later to the Hadoop Master. Figure 5.6 presents the process state of the three significant processes inside each VM. As shown, VM-Slave 1 reads the data for a while and then starts sorting. During the read phase, the CPU is mostly idle and waits for data. By contrast, the cvCPU of VM-Slave 2 does not wait for disk since it reads the data faster. Table 5.4 shows the percentage of wait for different resources ($W_{i,j}^{Disk}$, $W_{i,j}^{Net}$, $W_{i,j}^{Task}$, and $W_{i,j}^{Timer}$). From this data we can infer that the wait for disk ($W_{i,j}^{Disk}$) in

VM-Slave 1 is larger than in VM-Slave 2. Moreover, Figure 5.7 shows the frequency of wait for Disk (f^{Disk}) and Network ($f^{Network}$) for VM-Slave 1 and VM-Slave 2. VM-Slave 1 waits more frequently for the disk as compared to VM-Slave 2.

Figure 5.6 The Hadoop TeraSort algorithm is used to sort 50GB of Data, VM-Slave 1 is late because of waiting for Disk

The reason for this delay in VM-Slave 1 could be a low cap on disk usage, a slower disk, or contention with another disk intensive VM. Upon further investigation, we found that the problem was setting the disk cap too low.

There are several tools available to check the health of VMs. They poll instances at specific intervals and then mark them as *UNHEALTHY* if they do not respond successfully [119]. Furthermore, it identifies the action that the system administrator should take to solve the issue. For our internal host, we developed a lightweight Health Check tool that monitors the VMs disk performance. The metrics provided by our Health Check tool are: Disk Read/Write per second and wait time for Disk. Using the VM health check tool, a disk alert was received for one of our VMs saying that the wait for Disk is higher than expected. In our investigation with LTTng and ASD (shown in Figure 5.8), we found that the VM is frequently waiting for disk. Our first guess was checking the CPU cap for VM. Then, we analyzed other VMs to see if the VM is contending with others on same disk or not. Upon further investigation, and looking at Nested VM vCPU view, we found that the issue was coming from running the nested VM. Figure 5.9 presents the nested VM vCPU view. As shown, the issue was not in the VM itself. Indeed, the nested VM, not the VM, waits for disk. We solved the problem

Table 5.4 Wait analysis of Hadoop TeraSort

Processes	Root	non-Root	Task	Timer	Disk	Net
2039963648	0.004	1.499	54.040	43.512	0.198	0.217
869769216	0.001	0.979	39.938	56.978	1.251	0.700
2046287872	0.004	3.125	57.357	38.022	0.766	0.283
2029756416	0.002	1.898	36.508	60.883	0.098	0.340
877412352	0.001	0.970	24.947	72.767	0.029	1.130
886243328	0.005	8.350	85.240	5.588	0.003	0.258

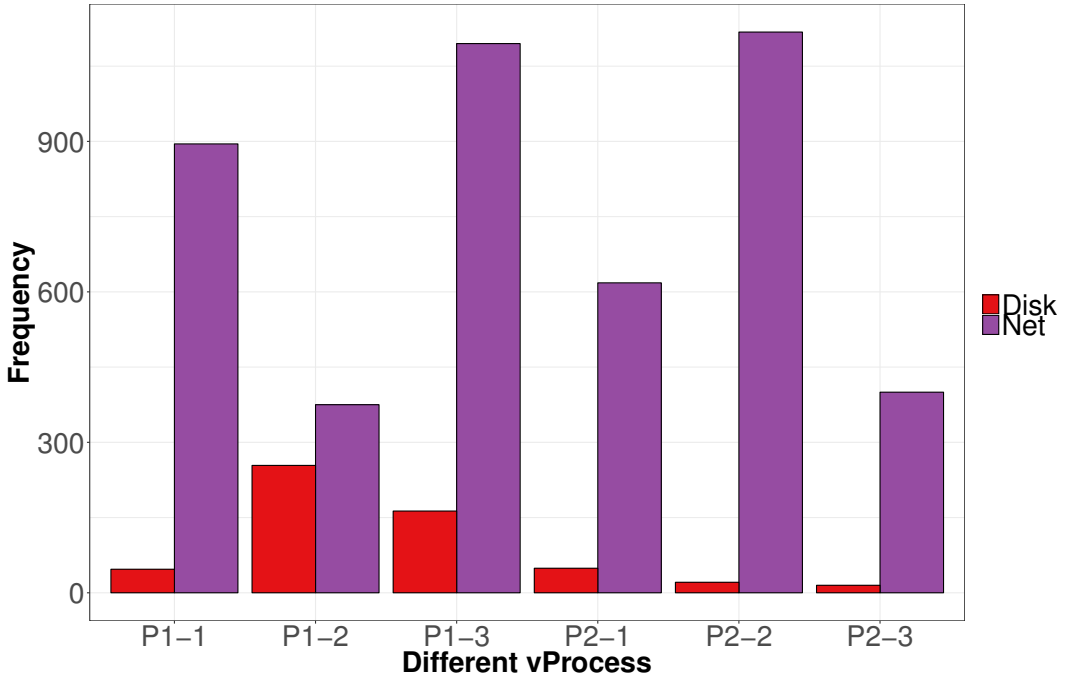


Figure 5.7 Frequency of wait for disk for Slave 1 and Slave 2 - P1-1: 2039963648, P1-2: 869769216, P1-3: 2046287872, P2-1: 2029756416, P2-2: 877412352, P2-3: 886243328

by changing the configuration of the nested VM.

In the next experiment, we removed all disk usage cap and put the VMs on a fast SSD to see a reduction in disk related performance degradation. Using our health check tool, we got another alert. Using GTA, we found that the two VMs are waiting for disk and there is contention between the two VMs in order to use the shared disk. Figure 5.10 shows the significant processes in the two VMs. As shown, the two VMs require disk accesses at the same time and wait for the completion. In order to solve this problem, we migrated one of the disk intensive VMs to another resource group.

5.6.2.2 Wait for Network Issue

The emerging Internet Of Things (IoT) has given rise to the concept of message-based Remote Procedure Call (RPC) microservices, to compute massive amounts of data. Message-based RPC microservices have no direct dependencies on other services, and the clients have no explicit knowledge of the respondent service. This complexifies the task for performance analysis tools, to detect issues in such a distributed platform.

In order to show how our technique can find issues in distributed platforms, we wrote a RPC client and server. The RPC client is put in VM2 and sends RPC requests to the server every

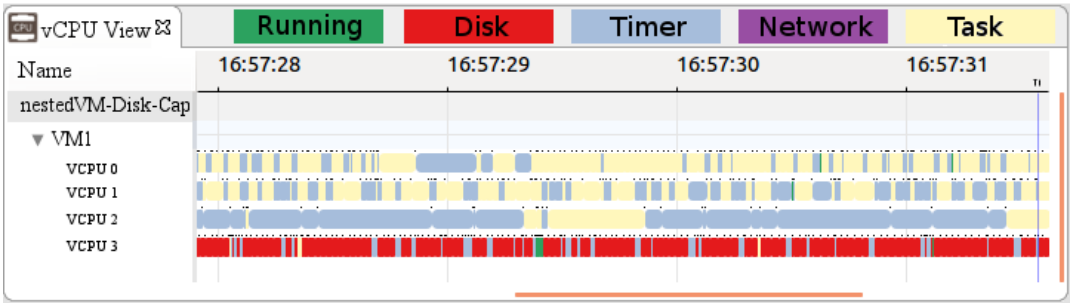


Figure 5.8 VM Disk

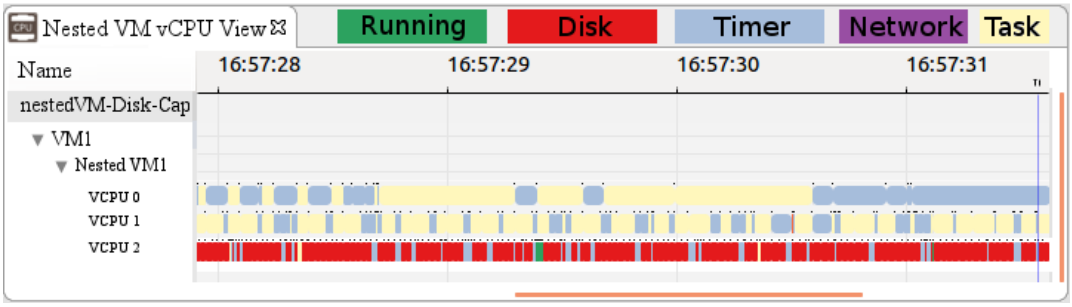


Figure 5.9 Nested VM Disk

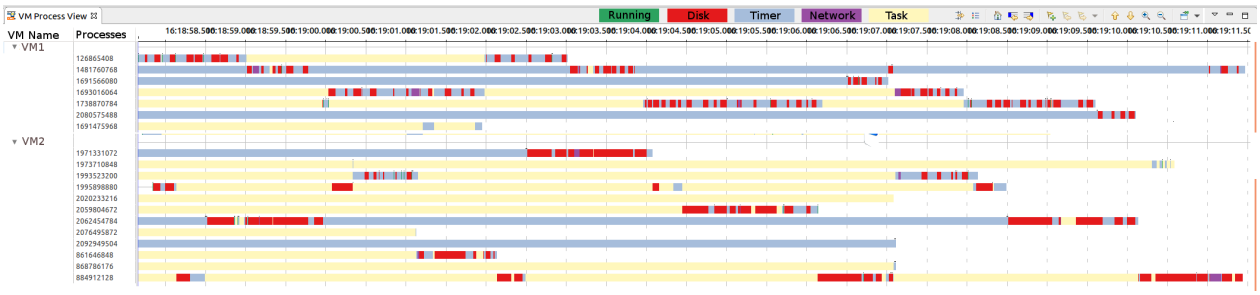


Figure 5.10 Wait analysis of vProcess when there is resource contention between two VMs

10ms. The command sent is a program to calculate Fibonacci numbers (as an example of a very simple service). The `tc` traffic shaper is used to manipulate the traffic control settings. It is applied to the virtual network interface to add network latencies of 30ms for the RPC server and 25ms for the RPC client. Figure 5.11 depicts the significant processes for both the RPC client and server VMs. The green intervals are the running state , the purple intervals are waiting for network, and the light blue intervals are waiting for timer. We also observed the effect of the network delay on the RPC server and client. The RPC client waits 55ms for the network to receive the response (25ms delay in the client network + 35ms delay in the

server network). The RPC server waits for 65ms to receive the new request from the RPC client (25ms delay in the client network + 10ms timer between each request + 35ms delay in the server network).

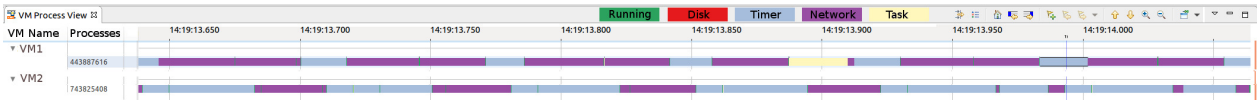


Figure 5.11 Wait analysis of RPC client and server with network issue using GTA algorithm

In another experiment, the RPC server and client are executed when there is a natural network latency. Since both VMs are on the same host, the latency between the two VMs is less than 1ms. Figure 5.12 depicts the state of vProcesses for both VMs. Also, we depict the critical path of the request in Figure 5.13. As shown, the RPC client receives the response from the RPC server immediately and does not wait for the network. The RPC server waits for 10 ms to receive the new request, since the RPC client sends a new request every 10 ms.

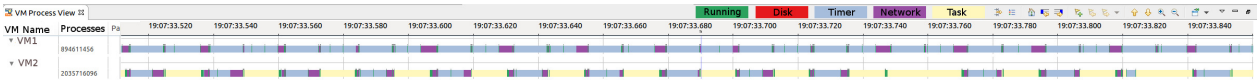


Figure 5.12 Wait analysis of RPC client and server without network issue using GTA algorithm

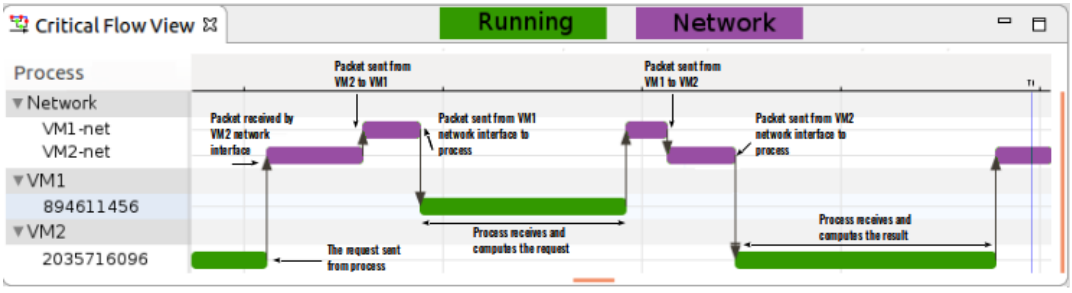


Figure 5.13 Critical Path of request, showing the interaction between VMs

5.6.2.3 Wait for CPU Issue

Many customers run CPU-intensive workloads in the cloud, often using parallel processing frameworks to distribute the work and collect result data. The CPU-intensive applications require high-end computing nodes to minimize the total execution cost and response time.

We experimented with CPU-intensive workloads and offloaded them to our internal private cloud. For security reasons, each workload executed in a separated nested VM. As shown in [114], the overhead of virtualization is not significant for CPU-intensive jobs, even inside nested VMs. However, we found that sometimes the execution time of the same task was much more than expected. We monitored the CPU-usage and process usage using the pre-installed system administration toolbox in Linux. All the tools show that the CPU is being fully utilized and there is no wait for resource. Then, the issue was investigated using the ASD algorithm. We found that VM1 and VM2 are preempting each other most of the time. Furthermore, by analysing the nested VMs individually, we observed that two nested VMs inside VM1 were also preempting each other. As mentioned before, preemption can happen in any virtualization level, causing latency for the applications inside VMs. Figure 5.14 depicts the graphical view of the ASD algorithm.

IaaS providers attempt to increase their profit by resource overcommitment while maintaining a high QoS. There is a direct trade-off between CPU overcommitment and preemption. Preemption is one of the most important factors in service level agreements (SLA). Using the ASD algorithm, the cloud provider can find out when/where preemption occurs for arbitrary nesting depths.

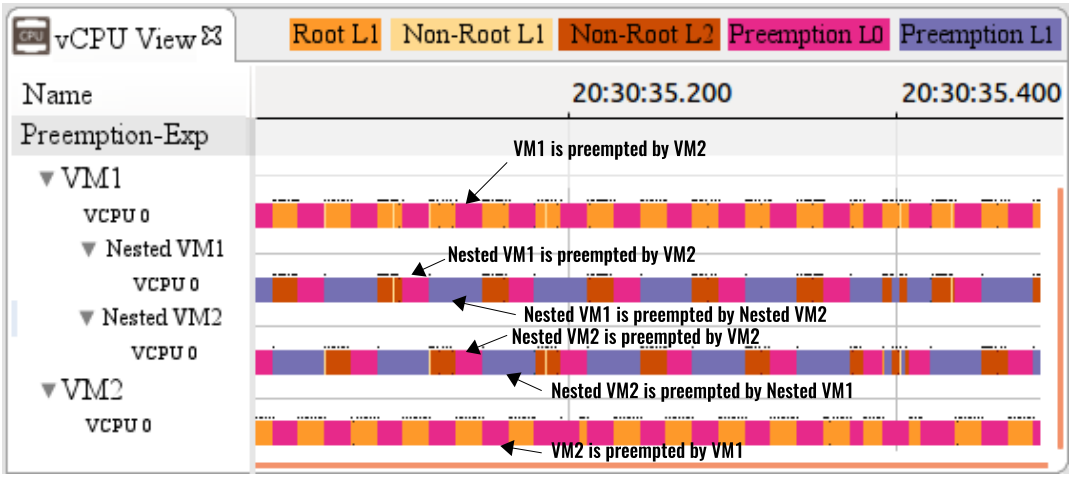


Figure 5.14 CPU preemption for different levels of virtualization

5.6.2.4 Wait for Memory Issue

Cloud providers increasingly offer to developers the choice of in-memory computing. In-memory computing keeps the information in the main RAM of dedicated VMs rather than in relational databases operating on slow mechanical disks. As a result, in-memory databases are

much faster than disk-optimized databases. Redis, Memcached and MemSQL are examples of in-memory computing applications on top of the cloud infrastructure. In this experiment, we show how our technique can identify waiting for memory in the cloud environment.

We deployed five VMs on the same host and executed a memory-intensive job on VM1, VM2, and VM3 to write random numbers into 1GB of memory. For the other two VMs, a CPU-intensive application was executed. In order to overcommit the memory, we developed another program that uses 25 GB of RAM in the host. The result of our experiment is found in Table 5.5. We found that the three first VMs, with memory-intensive jobs, suffer more from memory overcommitment. Our memory-intensive program was executed for 1.5s in average, in three VMs, but 15% of their time is wasted because of memory overcommitment. Also, we observed that VM3 suffers less from memory overcommitment. Our technique can also detect if the memory over-commitment happens at any level of virtualization.

Table 5.5 Execution time for different VMs when the host is suffering from Memory over-commitment.

VM name	*Execution Time(ms)*	*Freq EPT Violation*	*Violation EPT Time(ms)*	*Percentage(%)*
VM1	1329.0	3554	237.4	17.8
VM2	1834.5	18801	260.5	14.2
VM3	1332.4	15288	141.2	10.6
VM4	1169.1	0	0	0
VM5	1857.8	30	0.2	0

5.6.3 Use case 2: High Availability and Performing Rolling Updates

Failures may happen at different levels, but the system administrator should plan for those cases when something goes wrong. To offer a highly available service, it is imperative to have replicas. Replication can ensure that the service is almost always available. Cloud computing, with the availability of many nodes, helps supporting high availability at a low cost, in order to mitigate disasters. In this subsection, we analyse planned and unplanned downtime. Planned downtime is the time needed for upgrades or system maintenance, during which a system cannot be used to serve users. This is one example where node replication is useful to avoid downtime. Unplanned downtime is when the system cannot be used because of unforeseen failures in the hardware or software. This is also called breakdown maintenance and may be very expensive in terms of cost and time. In the case of breakdown maintenance, a load balancer may be used to first detect the failure and then transfer the request to a new node.

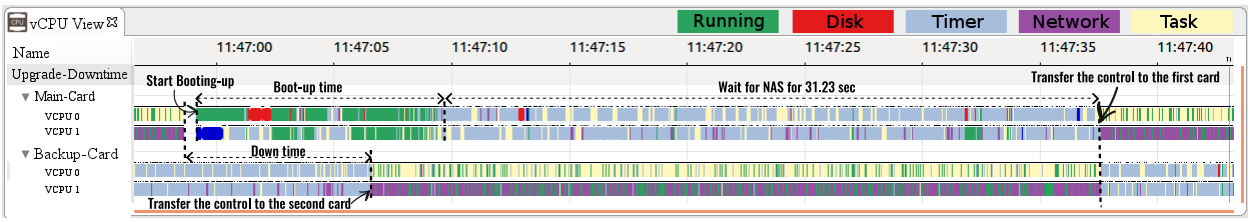


Figure 5.15 big downtime - Upgrade System downtime is 7.56 sec

A particularly interesting case was seen in a product, where some important downtime was encountered while there was one Card for serving the request and another Card as backup. We analysed this system during a breakdown maintenance. Figure 5.15 illustrates the sequence of events during a breakdown maintenance. In this experiment, Main-Card VM serves the incoming request and the Backup-Card VM is the replicate. HAproxy is being used as load balancer to monitor the service; in case of failure, it transfers the request to another Card. We traced the host hypervisor during a breakdown to analyse the behaviour of that configuration. We found that HAproxy perfectly detects that the Main-Card is down and transfers the request to the Backup-Card. Later, when the Main-Card is back up and available, it transfers the request to the Main-Card. However, we found that the service downtime was about 7.56 seconds. During this time, no request is served and they should all be transferred to the backup server. This overloads the backup server, which causes unexpected delays for user requests. The 7.56 seconds delay appears to be a default configuration of HAproxy for monitoring services. The default health check period could be reduced, in order to decrease the downtime, but this could add extra overhead.

Another interesting observation from Figure 5.15 is that for about 35 seconds the whole system becomes vulnerable and loses its redundancy. Upon further investigation, we realized that the requests were not transferred back to Main-Card immediately after VM boot up. We detected that the Main-Card service was not available even after being rebooted. Using the HEC algorithm, we found that the service was blocked for about 31 seconds, waiting for the `syslog` process. Indeed, the Main-Card VM attempted to offload the logs to an unavailable remote server, before timing out and resorting to another log server.

The next experiment is a rolling upgrade, a form of planned downtime. Recent studies [120], [121] show that software upgrades are the main cause of downtime, sometimes causing a failure of the whole system. Rolling upgrades is a best practice for software upgrades. With this technique, one node at a time is upgraded and rebooted. This reduces the downtime, since only a single node at a time is unavailable. However, it could result in a temporary

downtime and performance reduction, because of the unavailable node and the transfer of requests to the remaining nodes. Figure 5.16 illustrates the rolling upgrade sequence. As shown, the downtime is short (less than 100ms) compared to the unplanned maintenance case. A 100ms latency is acceptable, even for real time services like VoIP [122].

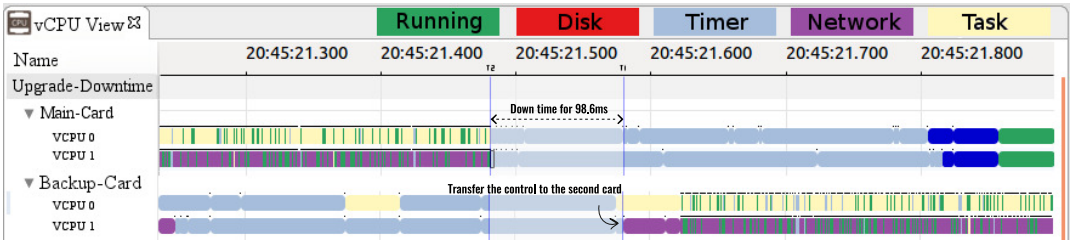


Figure 5.16 Small downtime - Upgrade System downtime is 98.6ms

5.6.4 Use case 3: Concurrent Processes

The design of a concurrent software system always involves several challenges. The key to parallelism is to keep all cores busy with useful computations at all time, thus no waiting/contention on resources or locks, and efficient load sharing/balancing. Service dependability is a major concern for programmers, especially if the services are executed on a VM. Blocking state detection is crucial to finding anomalous service behaviors that may lead to the violation of service level agreements (SLAs).

To achieve a high degree of parallelism, it is important to split a program into individual parts that do not need to communicate much with each other. As a result, the processes do not block for each other. When programs need to exchange information, in order to continue executing, they have to wait until the information becomes available. To efficiently use the available parallel hardware, we must avoid contention on shared resources, which can block some threads and waste processor time. The proposed HEC algorithm can determine if parallelism can make algorithms run faster in the cloud environment. It can also show the regions where individual threads in the program do block.

An application was designed to benefit from a high degree of parallelism. However, the resulting performance was lower than expected. The host was traced and the HEC algorithm executed to expose any issue. Figure 5.17 depicts the graphical view of the application. Here, tasks 29 and 30 could be executed concurrently after task 10. Task 10 blocks other processes and suspends them until its job gets finished. Then, the other tasks resume. In this experiment, the HEC algorithm, not only shows the dependencies between processes but also how processes are dependent on resources. As shown, process number 10 is a disk intensive

task and waits most of the time for disk. It is probably a service that analyzes some data and then writes them into a database, while other tasks will be locked out, until the whole data is written to disk. This issue could be solved using a faster disk drive like an SSD.

5.6.5 Use case 4: Communication Intensive VM

NGN (Next Generation Network) is a concept that has been introduced to take into account the changes in the telecommunications field. The core network of NGN is the IP Multimedia Subsystem (IMS), standardized by the 3rd Generation Partnership Project (3GPP), in order to offer deployment of new rich media communication services, mixing telecom and data services [123]. However, IMS is not fully implemented yet, because of the cost and complexity of the associated standards. In recent years, with the introduction of network virtualization, IMS has become more readily achievable [124].

Despite its merits, IMS still faces important challenges in terms of scalability and elasticity. Indeed, by using text-based, bandwidth-hungry, and delay-inducing protocols (such as SIP), and also with the constant increase in user demands, IMS nodes may quickly become overloaded, as shown in [125].

To analyse the behaviour of such a network, we deployed an IMS system with all its components (PCSCF, SCSCF, ICSCF, and HSS) in a VM. To simulate users, we used uctIMSclient to generate a load on the vIMS network. We found that sometimes the latency for setting up a call is more than expected. To resolve the issue, we traced the host and used the HEC algorithm. Figure 5.18 depicts the critical path for handling an Invite message in this dynamic vIMS network. As shown, when vIMS receives the Invite message, it queries the DNS server for the domain name (shown as wait for network). During this period, the vIMS waits for the response from the DNS server. This delay could be removed by increasing the TTL for the DNS entries. On the other hand, changing the TTL could affect the load balancer for this network. There is trade-off between decreasing the TTL and its effect on latency. After

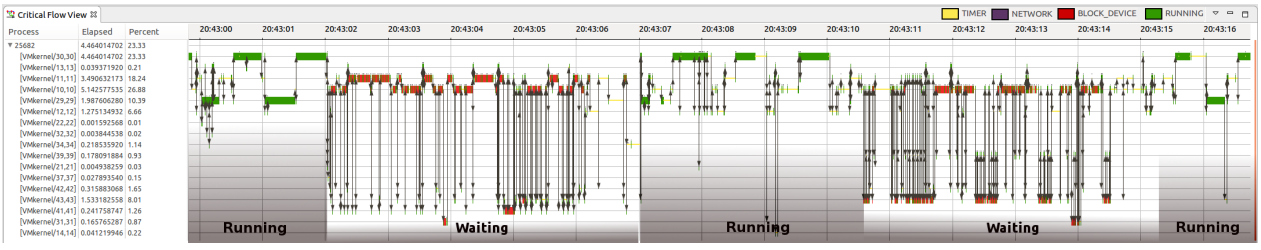


Figure 5.17 Analyzing a VM that has process

querying the domain name, the vIMS network exchanges some messages between the users and loads their profile from SCSCF. Then, at the end, the call will be established.

5.6.6 Use case 5: APT Analysis

The HEC algorithm is very useful for program comprehension. It helps software engineers and IaaS providers to better understand their application run-time behaviour and performance. Figure 5.19 depicts the critical path view of the Advance Packaging Tool (APT). APT is the official tool for Ubuntu and Debian users to update their software distribution. However, it is difficult to understand its performance because of all the pre and post install scripts that vary from one package to the next. We divided the analysis of the apt-get tool into four stages. In the first segment (label 1) of Figure 5.19, apt-get communicates with other processes to download and read the cached packages. In segment (2), apt-get interacts with the Debian package manager (dpkg). It installs the packages, along with the downloaded dependencies. In segment (3), numerous interactions with other processes are shown. Zooming to see the details, we can infer from this step that the VM waits for the block device for a while and then enters into this stage. We investigated more and found that it is related to the installation of man-pages. This step creates 340 mandb processes for about 1.4s. Creating a process is a relatively lengthy task, especially in a VM. To solve this issue, a thread pool could be used instead of creating new processes.

5.6.7 Use Case 6: Bootup analysis

Many new applications are based on micro-services. It makes them faster and more flexible, stable and modularized. To execute these tiny services, a VM or a container is created and later terminated. Creating and terminating a VM is a time-consuming task. Moreover, VM users sometimes experience a slow boot process (from VM power on until the VM is ready to serve users).

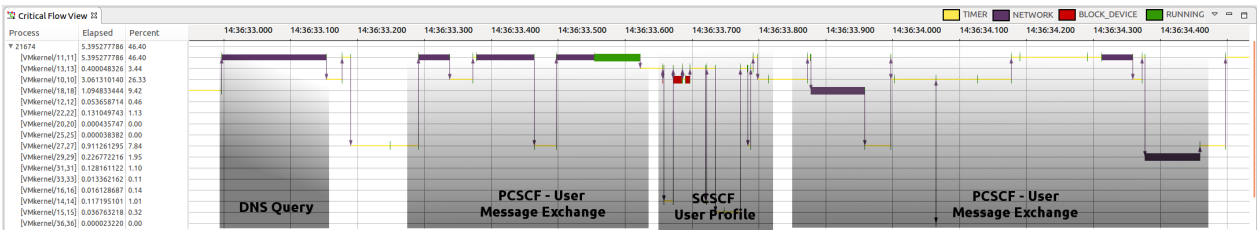


Figure 5.18 Latency analysis of handling Invite message using HEC algorithm

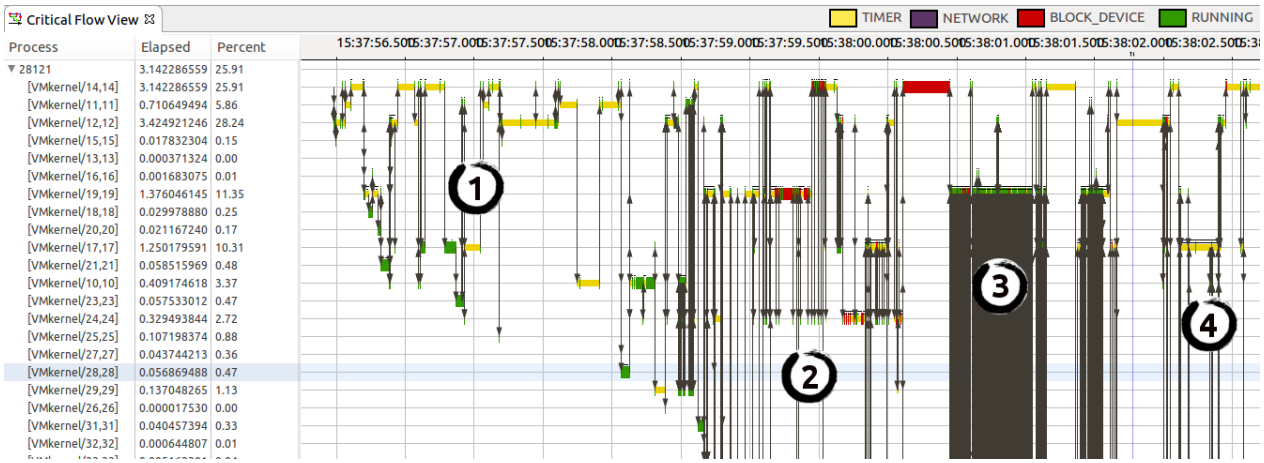


Figure 5.19 Reverse engineering of APT using HEC

In this subsection, Linux kernel 4.13, with different build targets, and Windows 10 Pro boot time are studied. Our VMs had 4 vCPUs and booted up separately. In the first experiment, the Windows boot up time is studied while running on a Hard Disk Drive (HDD). Then, we compared it with the boot up time of Windows on Solid-State Drive (SSD). As expected, on a SSD the Windows VM boots up faster because of the shorter disk wait. However, the CPU usage for both (SSD and HDD) is almost the same.

In the second experiment, Linux kernel 4.13 was compiled with the *allmodconfig* target. This configuration compiles all possible modules and creates a heavyweight Linux kernel. As expected, because of the big kernel with numerous modules probing the hardware, the boot-up time increases significantly. As Figure 5.20 depicts, the total boot-up time for this non-optimized kernel is 50.15s. The vCPU waits for the disk for 13.51s and waits for tasks for 49.55s (for 4 vCPUs). In all our experiments, there are some unknown states for vCPUs that our technique cannot resolve, because in the early stage of boot up (before *init* process), tracing is not fully operational.

The third experiment boots the Linux kernel with the *localmodconfig* build target. The *localmodconfig* target usually discovers the kernel minimal useful configuration and disables any module that is not being loaded. With this configuration, a lighter (semi-optimized) Linux kernel is created that boots up much faster. As the figure shows, the boot-up time, wait for tasks, and wait for disk (for total 4 vCPUs) are 15.6s, 16.66s, and 7.07s, respectively. Then, we moved the semi-optimized Linux kernel to a Solid State Drive (SSD). As shown, the VM boot up is faster (5.22s) because of the shorter disk wait.

With our technique and analysis, we compared the Windows and Linux boot up times. The

optimized Linux Kernel running on SSD boots up faster than any other operating system. The non-optimized Linux Kernel running on HDD boots up slowly. The optimized Linux Kernel running on SDD or HDD boots up faster than Windows. We also inferred that the boot-up time is highly dependent on the Block I/O device speed. Figure 5.20 depicts that more I/O waiting implies more waiting for tasks, and as a result more waiting for timer.

Using ASD, the IaaS provider can tune the allocated resources based on VM needs. They can find out the reasons for waiting and solve the issue by provisioning that specific resource.

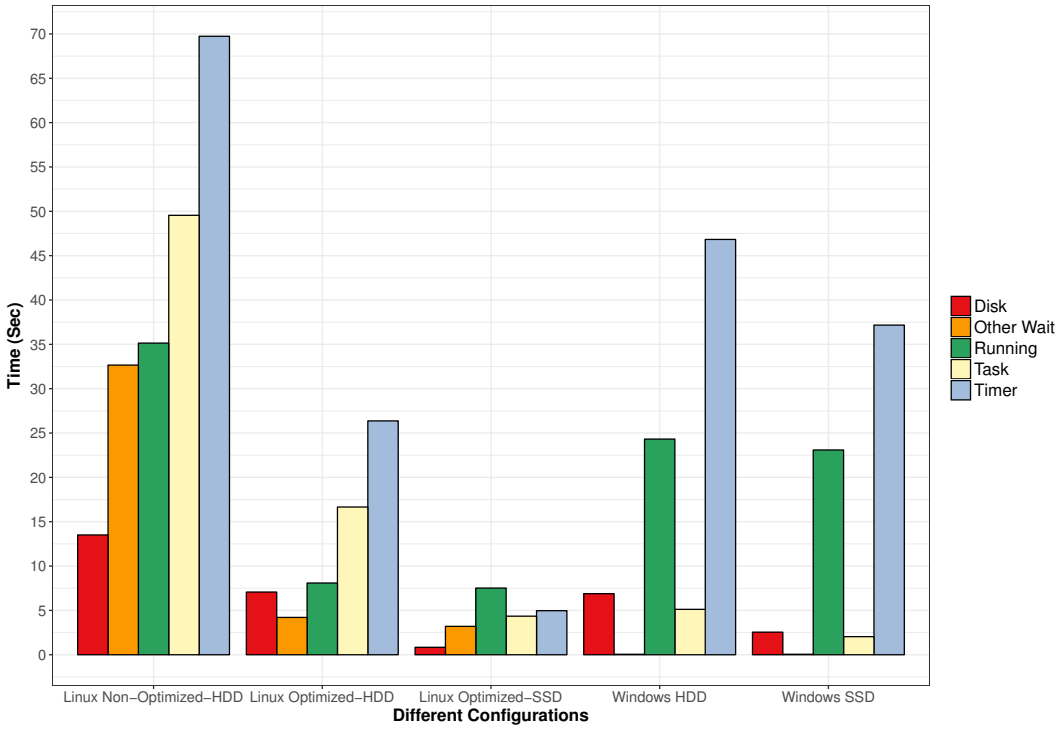


Figure 5.20 Bootup for Linux and Windows 10 Pro

5.6.8 UseCase 7: Sensitivity Analysis

Cloud providers now host thousand of VMs to fulfill the needs for cloud services and distributed applications. Co-locating VMs that communicate with each other can optimize the network performance but may decrease tremendously their QoS. It is important to understand the behaviour of VM workloads and to quantify their sensitivity to specific resources. Sensitivity analysis is performed by varying the resource caps over a range of values, and studying the effect on different metrics. With this technique, the cloud provider can detect resource configurations that deserve more attention for improving the QoS of VMs. In this subsection, we use tracing and the GTA algorithm to characterize the sensitivity of VMs (in

any level) to their allocated resources.

Some of the most popular application like Apache2 and MySQL were used to show how our technique can reveal the sensitivity of VMs to their resources. Apache2 is known as a network intensive application and MySQL is known as a disk intensive data-base management system. For Apache2, the `tc` command is used to add a delay to the virtual interface of the VM. Moreover, to benchmark the sensitivity of MySQL, a disk cap is put to limit the write and read bandwidth to 512 KB/s. Table 5.6 represents the total duration of each state, for the Apache2 and MySQL life-cycle. We can infer that Apache2 is highly sensitive to network latency but MySQL is not very sensitive to disk cap. The percentage of wait for network for Apache-0D (Apache with natural network latency) is about 1.5% as compared to 15% for Apache-50D (Apache with 50 ms network delay), and 22% for Apache-100D. Another interesting result is depicted in Table 5.7 which shows the frequency of wait for network and disk. We can see that Apache webserver is very sensitive to network latency. Apache-100D waits 16 times more than Apache-0D.

As we can infer from these two tables, suprisingly, MySQL is not as sensitive to disk as expected. MySQL-512, compared to MySQL-0, waits less for Disk. MySQL is more memory-bounded application.

5.7 Evaluation

5.7.1 CPA of Trace Compass vs. HEC

TraceCompass [2] provides a critical path graphical view proposed by [87] to follow the execution path of an arbitrary process. In this experiment, we illustrate the differences between our method (HEC) and the CPA of TraceCompass.

Our method reveals contention (as compared to the CPA of Trace Compass) on a shared resource, since it collects information about the VMs from the hypervisor level. As an

Table 5.6 Wait analysis of different applications, to find out the sensitivity of a VM to a specific resource

Processes	Root	non-Root	Task	Timer	Disk	Net
Apache-0	0.075	38.536	46.945	4.947	0.0	1.545
Apache-50	0.092	18.900	53.405	3.412	0.002	15.016
Apache-100	0.052	11.952	54.067	6.439	0.004	22.270
MySQL-0	0.257	85.393	11.395	2.908	0.0	0.045
MySQL-512	0.003	83.000	15.774	0.873	0.004	0.0

Table 5.7 Frequency of wait for different applications, to find out the sensitivity of a VM to a specific resource

Processes	f^{Task}	f^{Timer}	f^{Disk}	$f^{Network}$	f^{Total}
Apache-0	13161	108	0	586	13855
Apache-50	96667	4764	4	79654	181090
Apache-100	99078	10257	11	118307	227654
MySQL-0	694	62	0	3	759
MySQL-512	393	40	1	0	434

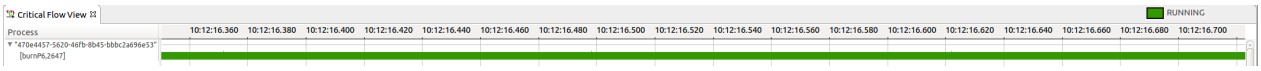


Figure 5.21 Analyzing CPU-burn using CPA algorithm

experiment, we compared these two techniques when there is contention on pCPU. We used a VM with one vCPU and executed *CPU-burn* (a pure CPU contender, 100% using a CPU) on it. We then traced the VM and the host at the same time using LTTng. Figure 5.21 shows the result of using the CPA on the VM trace. As we can see, there is one process running inside the VM and the CPA shows that is executed all the time. We used the HEC algorithm on the trace taken from the host, as shown in Figure 5.22. It shows that the VM was not running all the time, and it shared the pCPU with other processes running in the Host.

5.7.2 Overhead Analysis

In this subsection, we conducted experiments to evaluate the tracing overhead of HEC. Then, we compared it with the CPA algorithm, as proposed in [87]. The tracing cost consists of tracepoint instructions in the code path, and the tracing consumer daemon, which is responsible for offloading the events from in-memory buffers to disk. It is crucial to understand the effect of tracing on our system. Having a low overhead method ensures that the system behaviour, with tracing enabled, is similar to that of the original untraced system. To do so, for this use case, we measure the overhead incurred by tracing the host only (HEC), as well

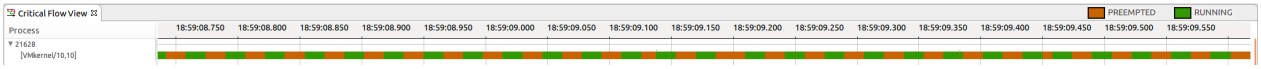


Figure 5.22 Analyzing CPU-burn using HEC algorithm

as the VM only (CPA).

In order to compare the two approaches, we enabled the tracepoints needed for the CPA approach, and traced the VMs. As shown in Table 6.4, the critical path analysis approach adds more overhead through all tests, since it needs to trace the VMs. We used the Sysbench benchmarks to measure the overhead of both approaches, with Sysbench configured for Memory, Disk I/O and CPU intensive evaluations. The scheduler governor of Linux was set to `performance`, to remove the variations due to CPU frequency changes. We executed the same test several times, with and without tracing enabled. Then, we compared the execution time of each test and computed the overhead.

Our approach has negligible overhead for CPU and Memory intensive tasks, at 0.3% or less. The average CPU usage for the tracing consumer daemon is between 0.8 and 8.2 percent of one CPU core, depending on the event production rate [87]. Our new HEC algorithm needs only to trace the host, so all tracing tool daemons are decoupled from VMs and do not affect the system performance.

5.7.3 Ease of Deployment

The ASD, GTA, and HEC algorithms use only 4 host hypervisor tracepoints. Other available methods for VM analysis ([108] and [73]), need to enable most tracepoints in VMs and the host. In the absence of a global clock, between the host and each VM, more tracepoints must be enabled for a precise time synchronization. This adds extra overhead to the VMs and host. It also increases the computation time for the analysis part. Our method does not need time synchronization, since it receives all the events from a single source, the host. Furthermore, it can analyse the behaviour of other OSs in VMs (Linux, Windows, and Mac OS), since it only needs to enable the events illustrated in Table 5.3 on the Linux host.

Table 5.8 Overhead Analysis of HEC as compared to CPA

Benchmark	Baseline	CPA	HEC	Overhead	
				CPA	HEC
File I/O (ms)	450.92	480.38	451.08	6.13%	0.03%
Memory (ms)	612.27	615.23	614.66	4.81%	0.01%
CPU (ms)	324.92	337.26	325.91	3.65%	0.30%

5.7.4 Limitations

Our technique receives its events from the host. Thus, guest OS specific information (like process name, PID, filename, etc.,) is not accessible with our method. For example, our method cannot extract information about containers in a VM, since it does not involve virtualization, but it can show all the threads and processes (whether or not containers are used) using the GTA algorithm. The other trace-based methods ([108] and [73]) provide more useful insights about the processes and their interactions with the guest kernel.

5.8 Conclusion

Service virtualization in the cloud is very common. The diagnosis of performance degradations in cloud environments is thus extremely important but was a significant challenge. This has become an important field of research and numerous tools have been proposed to address the problem of debugging and troubleshooting VMs. However, none of the available techniques provided a clear picture of the execution graph and dependencies between processes in VMs.

In this paper, we proposed a method to analyse the execution of a distributed and hierarchical virtualized environment, using scheduling, network and virtual interrupt events. Our tracing and analysis technique does not require internal access to the VMs, which is often not available due to security and overhead implications. Furthermore, our techniques can measure the dependency on various resources. The HEC, GTA, and ASD algorithms answer the typical question of where lies the bottleneck. Our benchmarks show that the overhead for our approach is around 0.3%. By contrast, the overhead of other approaches ranged from 3.65% to 6.13%. Moreover, our approach provides more complete information since it has access to host hypervisor level information. As future work, we would like to automatically detect bottleneck segments using Machine Learning algorithms.

CHAPTER 6 ARTICLE 3: "HOST-BASED VIRTUAL MACHINE WORKLOAD CHARACTERIZATION USING HYPERVISOR TRACE MINING"

Authors

Hani Nemati, Seyed Vahid Azhari, Mahsa Shakeri, and Michel Dagenais

Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

Submitted in ACM Transactions on Modeling and Performance Evaluation of Computing Systems

Reference as Hani Nemati, Seyed Vahid Azhari, Mahsa Shakeri, and Michel R. Dagenais, "Host Hypervisor Trace Mining for Virtual Machine Workload Characterization," ACM Transactions on Modeling and Performance Evaluation of Computing Systems, Under-Review

6.1 Abstract

Cloud computing is an emerging technology that provides on-demand access to a pool of shared resources. Such distributed and complex environment requires advanced resource management solutions which could model virtual machines (VM) behavior. Different workload measurements, such as CPU, memory, disk, and network usage, are usually derived from each VM to model resource utilization and group similar VMs. However, these course workload metrics require internal access to each VM, which is not feasible with many cloud environments privacy policies.

In this paper, we propose a non-intrusive host-based virtual machine workload characterization using hypervisor tracing. VM blockings duration, along with virtual interrupt injection rates, are derived as features to reveal multiple levels of resource intensiveness. In addition, the VM exit reason is considered, as well as the resource contention rate due to the host and other VMs. Moreover, the processes and threads preemption rates in each VM are extracted using the collected tracing logs. Our proposed approach further improves the selected features by exploiting a page ranking based algorithm to filter non-important processes running on each VM. Once the metric features are defined, a two stage VM clustering technique is employed to perform both coarse and fine grain workload characterization. The inter-cluster and intra-cluster similarity metrics of the silhouette score is used to reveal distinct VM workload groups, as well as the ones with significant overlap. The proposed framework can provide a

detailed vision of the underlying behavior of the running VMs. This can assist infrastructure administrators in efficient resource management, as well as root cause analysis.

6.2 Introduction

Recently, cloud computing technology has revolutionized software development and deployment at enterprises and organizations. The cloud environment provides on-demand access to shared resources, without the requirement of direct active management by the users. Despite the flexibility brought by this infrastructure, the complexity of such environments makes them prone to performance anomalies. For instance, an IaaS (Infrastructure as a Service) provider could include hundreds of hosts and thousands of Virtual Machines (VMs), which requires a complex distributed resource management. In such environment, an anomaly in one VM might cause performance degradations in other VMs, which could lead to significant financial penalties. In this regard, effective automatic monitoring of resource workloads could aid the service administrators to determine the VMs behavior and thus avoid likely failures.

We argue that an effective approach to reduce the complexity of VMs monitoring is to leverage a VM clustering technique. In this way, VMs are grouped into separate clusters, where VMs with similar behavior are categorized together. Therefore, the service administrator deals with a group of clusters rather than thousands of VMs, which facilitates the resource management procedure. For instance, consider the case where there is an excessive load on a specific system resource. The clustering technique can group all VMs suffering from contention in the same cluster. As a result, the administrator can easily identify this anomalous cluster and resolve this issue by adding more resources or migrating some VMs to another host.

In order to perform the clustering technique in such an environment, monitoring metrics must be gathered from each VM. This is challenging since the infrastructure provider usually does not have internal access to each VM, because of security issues. In this regard, developing an agent-less monitoring technique is imperative to enable extracting metrics and features, without violating the VMs privacy policy.

In this paper, we propose an unsupervised clustering approach based on an agent-less feature extraction technique. No prior knowledge is considered about the type of applications and processes running on the VMs. The designed clustering approach enables categorizing the VMs based on their similarity in terms of execution workload. The features are extracted based on hypervisor tracing, using the existing static kernel and KVM tracepoints, along with an additional new dynamic tracepoint. Each virtual CPU (vCPU) state is analyzed as

well as the virtual interrupt injection rate. In addition, the VM exit reason is considered, along with the resource contention rate caused by the host and other VMs. In this work, we propose two approaches for feature extraction. In the first approach, features are extracted from the vCPUs of the VMs. In this case, we cluster VMs based on the behaviour of all their processes. The second approach uses the PageRank algorithm to model the importance of processes running in each VM. Then, features are extracted only from top processes instead of the vCPU. The performance metrics of the top important processes are then fed into a two phase clustering scheme based on K-Means. Inter-clustering and intra-clustering similarity metrics of the silhouette score is used to depict the distinct workload groups. Our proposed framework can be used by an IaaS administrator to improve scalability, resource management, and VM root cause analysis.

The main contributions of this work are: **First,** the feature extraction is performed in an agent-less manner, in which tracing and analysis are performed without internal access to the VMs. This is a critical point, since access to the VMs is usually restricted in a public cloud. **Second,** we propose a two phase feature selection and clustering technique which enables both coarse and fine grain workload characterization. **Third,** we go even further in the analysis by identifying the top important processes running on each VM, using a page ranking algorithm. This puts the irrelevant processes aside and thus provides a more distinct VM workload grouping. **Forth,** we evaluated our proposed approach on a real dataset, including different software applications (e.g., mySQL, Apache, etc.) and by considering various scenarios such as overcommitment of resources and other possible problems. Our database is available online to be used in future studies and to the benefit of other developers [126]. In addition, we implemented different graphical views as follows: a) VM similarity plot which depicts the level of similarity between VMs; b) a vCPU graphical view that presents a timeline for each vCPU and depicts different states of a VM for further analysis, after VM workload grouping. c) A process connectivity graph, which shows the communication between significant processes.

The rest of this paper is organized as follows: Section 6.3 presents previous approaches on VM feature extraction and clustering. Section 6.4 introduces some background about virtualization technology and presents the different states of the processes running inside the VMs and their requirements. In addition, it describes the algorithm used to detect different states of the vCPUs, along with our feature extraction approach. Section 6.5 explains the details of the VM workload clustering model. Section 6.6 presents our experimental results along with a discussion on the essential behavior of different clusters. In Section 6.7, we compare these approaches in terms of overhead, ease of use, and limitations. Section 6.10 concludes the paper with possible future directions.

6.3 Related Work

In this section, we present the available techniques for extracting workload-related metrics and survey the existing VM performance monitoring approaches in cloud infrastructures.

The VM workload metrics could be collected in two ways: agent-based and agent-less. The agent-based techniques execute a daemon inside VMs to gather and send metrics to an external module for detailed analysis. Many available approaches [94] [68] monitor running applications by injecting an agent into a VM. The agent usually relies on common Linux APIs or simply employs the existing monitoring tools such as `iostat` [79] and `vmstat` [80].

On the other hand, the agent-less techniques gather metrics without the requirement to have an internal access to the VMs. One example of such technique is the Virtual Machine Introspection (VMI) approach [83] which analyzes the VM memory space. Some studies like [84] and [85] have used the VMI technique for analyzing the performance of the VMs. Another approach to collect metrics in an agent-less manner has been proposed in [114], [116], and [109], where hypervisor level tracing analysis was employed.

Canali *et al.* [88] exploited the correlation between the usage of multiple resources to determine which VMs were following the same behavioral pattern. Then, the K-Means approach was applied to cluster together similar VMs in an IaaS cloud data center. Their method sampled resource utilization metrics, such as CPU, memory, disk, and network usage, with an assumption of no knowledge on the processes running on the VMs.

In [89] and [90], the Smoothing Histogram-based clustering (SH-based clustering) approach was proposed, in order to group VMs showing similar behavior in a cloud environment. The monitoring metrics were periodically collected from the VM resource usage using the hypervisor APIs. The Bhattacharyya distance has been applied to measures the similarity between two VMs based on the probability distributions of resource usage. A spectral clustering based approach was employed to categorize VMs into distinct groups. Their work focused on monitoring scalability by selecting few representative VMs as the VMs closest to the clusters centroids. Further, in [91] the authors employed a similar strategy to propose a VM placement technique, namely class-based placement. The class-based method leverages the knowledge of VM classes to select a few representatives and reduce the size of the problem to be solved. The solution obtained was then replicated as a building block to solve the global VM placement problem.

In another work [92], a K-Means clustering based approach was proposed to monitor the cloud security against various anomalies, like intensive resource usage and malware attacks. System-level metrics, such as CPU, Memory, Disk, and network usage were collected for every

monitored VM. In [93], the authors included the fine-grained information of each process in a VM for malware detection in cloud infrastructures. The proposed method trained a 2D Convolutional Neural Networks (CNNs) model on performance metrics gathered from processes running in the VMs.

Zhang *et al.* in [94] proposed a density-based clustering method (DBSCAN) for abnormal behavior detection in large scale clouds. The performance related metrics such as CPU usage, memory utilization, and disk usage were collected by BOSH agents. An entropy-based feature selection technique was used to reduce the data dimensionality.

All the studies examined so far extracted system level metrics, from process monitoring metrics (e.g., thread pool size) to resource utilization ones (e.g., CPU, memory, disk, or network usage). Other approaches have gone further by using lightweight tracing tools to gather detailed information on the underlying kernel and user-space executions.

In [95], the authors collected kernel event traces of all active processes in a cloud infrastructure and then pushed them into a central log store. An event sketch modeling module converted the raw event traces to a group of kernel events having causality relationships. A set of statistical and data mining analysis tools were offered to summarize the event sketches and perform cloud performance diagnosis.

Tracing VMs was used in [96] to detect Denial of Service (DoS) attacks in cloud infrastructures. A trace abstractor created high-level features from the statistics of the low-level events, such as CPU usage and the number of HTTP connections. The Support Vector Machine (SVM) algorithm was then applied to detect changes in VM behaviors. Another work in [68] used kernel level tracing for anomaly detection in cloud services. The proposed method, namely Perfcompass, was a statistical approach for finding external and internal faults in an online IaaS. PerfCompass was able to identify the top affected system calls and provide useful diagnostic hints for detailed performance debugging.

As mentioned in this section, none of the available cloud diagnosis approaches applied an agent-less technique with a lightweight tracing strategy. We believe that using detailed low-level information gathered by a lightweight tracer, in an agent-less manner, and clustering VMs with similar behavior could improve VM workload analysis. This improvement would be in terms of both security (agent-less technique) and overhead (lightweight Kernel level tracing).

Early results of this work have been presented in [127], which introduces an agent-less performance analysis approach to group virtual machines based on their workloads. In the current paper, this is expanded, notably by exploiting the PageRank algorithm (initially proposed for

website ranking) for VM workload analysis. This enables us to do performance analysis by considering only the most significant processes running on the VMs. Our currently proposed strategy reveals more detailed information regarding the VMs resource utilization and provides more distinct VM clusters. In addition, our proposed method requires to collect data at the hypervisor level, without an internal access to the VMs. Therefore, our approach needs a simpler tracing infrastructure as compared to other solutions, and adds less virtualization overhead.

6.4 VM Metrics and Features Extraction

This section first presents some common definitions and terminology in the world of virtual machines, along with some background information about virtualization. Then, the Wakeup Reason Analysis Algorithm (WRA) and Process Ranking Algorithm (PRA) are defined. The WRA algorithm is designed to detect different vCPU states in VMs and collect metrics for VM analysis. The PRA algorithm is developed to find the significant processes inside a VM by adopting a page ranking algorithm.

6.4.1 VMX Operation

Virtualization is supported by both Intel and AMD processors as the Virtual Machine extensions (VMX) and Secure Virtual Machine (SVM) instructions sets, respectively. Two types of VMX operation exist: **VMX root** and **VMX non-root**. Non-privileged instructions are executed as non-root while privileged instructions are executed as root mode. Therefore, when a privileged instruction comes, the control is passed to the Virtual Machine Monitor (VMM) in order to execute the instruction in a safe environment. After handling the privileged instruction, the VM resumes to VMX non-root mode. The transition between VMX root to VMX non-root is called a VM entry, whereas the transition between VMX non-root to VMX root is called a VM exit. The VMM monitors these transitions by updating an in-memory Virtual Machine Control Structure (VMCS). The VMM creates different VMCS for each vCPU and uses `VMREAD`, `VMWRITE` and `VMCLEAR` instructions to update VM informations. Each VM exit comes with an exit reason number, which provides useful information about applications running inside a VM. The VMCS structure contains several important fields that are used in this paper including: **CR3**: points to the page directory of a process in VM. **SP**: points to the stack of the thread inside the VM. Therefore, CR3 and SP can uniquely identify a process or a thread, respectively.

6.4.2 Virtual Interrupt Injection

Virtual Machine Extensions (VMX) instructions on x86 processors support virtual interrupts (`virq`) injection into VMs by filling the VM-entry interrupt-information field in the Virtual Machine Control Structure (VMCS). Interrupt injection includes software interrupts, internal interrupts, and external interrupts. Then, the processor uses the interrupt information field to deliver a virtual interrupt through the VM Interrupt Descriptor Table (IDT). Usually, the virq will be injected into the guest during the next VM exit by emulating the `LAPIC` register. Once the VM resumed, the pending interrupt is executed by looking up the VM `IDT`. After handling the interrupt, during the next VM exit, the pocessor performs an End Of Interrupt `EOI` virulization to update the guest interrupt status. Looking at the injected interrupt reveals useful information about the application running inside the VM. We elaborate more on the extracted metrics from the injected interrupt `virq` in the next subsection.

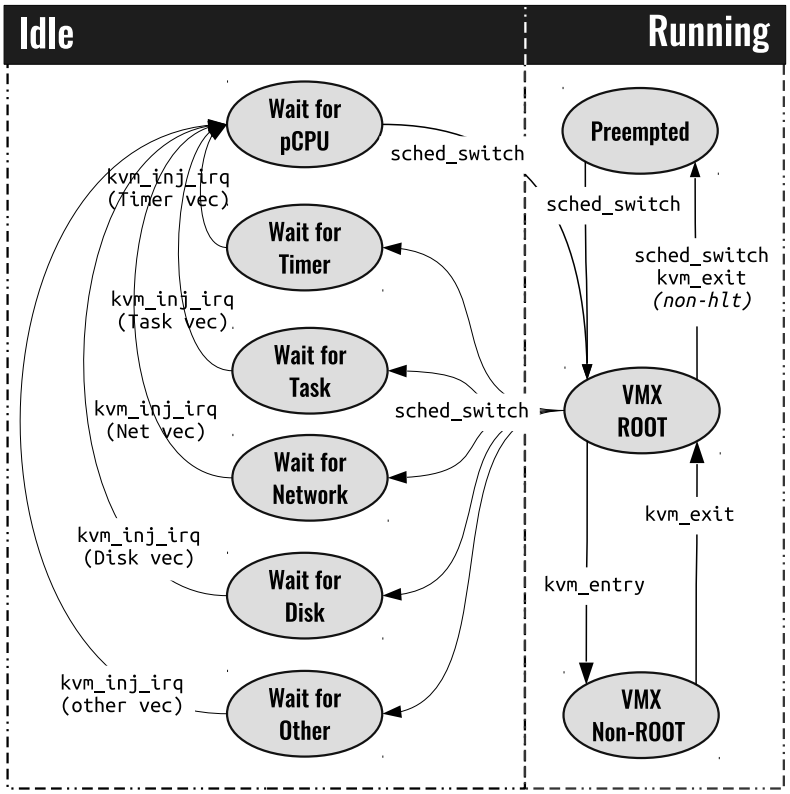


Figure 6.1 Virtual machine's virtual CPU state transition

6.4.3 VM Machine States Analysis

Different states of a vCPU on a VM and the conditions to reach them are shown in Figure 6.1. Each vCPU, similar to a physical CPU (pCPU), could be either in a running or idle

state. As mentioned in previous sections, a VM exits to the VMX root mode to execute a privileged instruction and then resumes to the VM non-root mode with VM entry. From a VM point of view, the whole resources are allocated to that VM, while in reality each VM shares the resources with other VMs and the host. This causes resource contention and leads to a preemption state. The preemption state occurs when the host scheduler takes the pCPU and gives it to another VM or process. In this state, the executing VM is halted for a while, without being notified. As a result, the VM process is in the running state inside the VM but does not execute any instruction. The preemption state is one of the states that cannot not be shown from inside a VM. The Idle state occurs when a vCPU is being scheduled out voluntarily by sending a `hlt` signal, which is a privileged instruction. This instruction causes an exit to the VMX root mode and notifies the host scheduler to schedule the vCPU thread out of the pCPU. In this state, the vCPU waits for a wake up signal from the host scheduler. Generally, the following main reasons cause the host scheduler to wake up the vCPU thread: **a)** Timer interrupt, a process inside the VM sets a timer and the timer is fired; **b)** IPI interrupt, inter processor communication wakes up a process; **c)** Network interrupt, a process inside the VM waits for an incoming packet from a remote process; **d)** Disk interrupt, a process inside the VM waits for disk. In addition to the Timer, IPI, Network, and Disk interrupts, the VM could wait for another device with different virq number. We label this state as "wait for other". Thus, the wait for X states as well as the running state will be used in our feature extraction technique to collect information about VMs.

We propose the Wake-up Reason Analysis (WRA) algorithm to detect the reason for waking up. Algorithm 9 shows the pseudocode for the WRA algorithm. The input to the algorithm is a sequence of events and the output is the detected wait for vCPU reason of a given VM. Whenever a VM has something to execute, it asks for the pCPU from the host scheduler. Receiving a `sched_wakeup` event shows that the VM asked for a pCPU and thus the state of the vCPU is updated to waiting for pCPU. Event `sched_in` shows that a vCPU is scheduled on a pCPU, but it is still in a waiting state. We call this state as wait for vCPU. Before running the vCPU on a pCPU, the hypervisor injects interrupts into the VM. These interrupts are the key in our analysis. The event related to interrupt injection is `vm_inj_virq`. The vec field in this event is compared with the Task, Timer, Disk, and Network interrupt number. The reason for waiting is updated based on the *vec* number. The different interrupt numbers are retrieved by following the pattern of using devices. This pattern is different between hypervisors. The `vcpu_enter_guest` event changes the vCPU state to running (Line 20). We also store the CR3 value in the payload of this event. CR3 points to the page directory of a process, in any level of virtualization, and can be used as an unique identifier for a process. As mentioned before, a running vCPU can be preempted either in host level (L0)

or guest level (L1). If the last exit is `hlt`, which means that the VM runs the idle thread, the state will be modified as wait for reason. In the other case of exit number, the state will be changed to preempted (Line 5). The preemption of L1 happens when the process inside the VM is preempted by the guest scheduler. In this case the CR3 will be changed without running the `hlt` instruction (Line 19).

6.4.4 Process Ranking Algorithm

In order to find the significant processes inside a VM, we adopt the PageRank (PR) algorithm [**?**] which has been used by the Google search engine to select significant web pages. The PR algorithm counts the number of links to a web page to evaluate the importance of that page.

Let p be a process and Bp a set of processes that point to process p. Then, let w be a process and Nw the number of links from w. Finally, let c be a normalization factor for the total rank of all processes. Then the rank (R) of p is computed as:

$$R(p) = c \sum_{w \in B_p} \frac{R(w)}{N_w} \tag{6.1}$$

The pseudocode for the Process Ranking Algorithm (PRA) is presented in Algorithm 10. It mainly uses the `sched_ttwu` event to identify the callee and caller processes (Line 5). Then, in cases where the injected interrupt is `IPI` (Line 9), it creates a link between the two processes (Line 11). Next, the PRA algorithm computes the rank of each process using our adopted page ranking approach (Line 13). Then, a breadth first search algorithm is used to find connected sub-graphs, showing different groups of processes. Here, the processes are grouped together if they wake up each other at least once. The de-noise function in line 15, receives the connected groups of processes along with their rank, and removes the groups that does not contain significant processes. In this work, we aggregate the metrics of the groups of processes that contain the top five significant processes.

For example, using events from Table 5.1, for c = 1 the page rank for different processes is computed as: R(P#1)= 2, R(P#2)= 1, R(P#3)= 1, R(P#4)= 1, and R(P#5)= 1. As a result, P#1 is considered as a very important process. Then, we use the breadth first search algorithm [**?**] to find the connected processes. Figure 6.2 depicts different groups of processes in our example. As shown, there are three groups and the g#1 group contains the significant process P#1.

To better understand the relationships between detecting significant processes and the PageRank algorithm, we provide an example. Here, we consider a simple algorithm to detect the

Algorithm 9: Wakeup Reason Analysis algorithm (WRA)

```
1  if event == sched_in then
2  |  vCPU_j^State = vCPU-root;
3  |  last_exit = getLastExit(vCPU_j) ;
4  |  if last_exit != hlt then
5  |  |  vCPU_k^State = Preempted_L0
6  |  else
7  |  |  vCPU_k^State = waiting_for_reason
8  else if event == vm_exit then
9  |  pCR3 = getLastCR3(vCPU_k);
10 |  putLastExit(exit_number);
11 |  vCPU_j^State = vCPU-root ;
12 |  pCR3^State = vCPU-root ;
13 else if event == sched_wakeup then
14 |  vCPU_j^State = pCPU-waiting ;
15 else if event == vm_entery then
16 |  lastCR3 = getLastCR3(vCPU_j);
17 |  if lastCR3 != cr3 then
18 |  |  vCPU_j^internal = Preempted_L1;
19 |  |  lastCR3^State = Preempted_L1 ;
20 |  vCPU_j^State = vCPU-non-root ;
21 |  cr3^State = vCPU-non-root ;
22 |  putLastCR3(cr3) ;
23 else if event == sched_out then
24 |  vCPU_j^State = idle ;
25 |  pCR3^State = idle ;
26 else if event == vm_inj_virq then
27 |  pCR3 = getLastCR3(vCPU_k);
28 |  if vec == Task Interrupt then
29 |  |  Update State for vCPU_j^State to Task;
30 |  |  Update State for pCR3^State to Task;
31 |  else if vec == Timer Interrupt then
32 |  |  Update State for vCPU_j^State to Timer;
33 |  |  Update State for pCR3^State to Timer;
34 |  else if vec == Disk Interrupt then
35 |  |  Update State for vCPU_j^State to Disk;
36 |  |  Update State for pCR3^State to Disk;
37 |  else if vec == Network Interrupt then
38 |  |  Update State for vCPU_j^State to Network;
39 |  |  Update State for pCR3^State to Network;
40 |  else
41 |  |  Update State for vCPU_j^State to Other Update State for pCR3^State to Other;
```

Table 6.1 Events and their payload based on Host kernel tracing
($virq\#1 = Timer, virq\#2 = Task, virq\#3 = Disk, virq\#4 = Network$)
sched_in_vcpu(vcpu, virq, cr3) = sched_in(vcpu) , vm_inj_virq(virq), vm_entry(cr3)
sched_out_vcpu(vcpu, virq, cr3) = vm_exit(reason), sched_out(vcpu)

#	Events	#	Events
1	sched_in_vcpu(*vcpu2, virq#1, P#4*)	22	sched_in_vcpu(*vcpu1, virq#1, P#5*)
2	sched_out_vcpu(*vcpu2, reason = hlt, P#4*)	23	sched_out_vcpu(*vcpu1, reason = hlt, P#5*)
3	sched_in_vcpu(*vcpu1, virq#1, P#5*)	24	sched_out_vcpu(*vcpu0, reason = hlt, P#1*)
4	sched_out_vcpu(*vcpu0, reason = hlt, P#1*)	25	sched_in_vcpu(*vcpu1, virq#4, P#3*)
5	sched_out_vcpu(*vcpu1, reason = hlt, P#5*)	26	sched_in_vcpu(*vcpu2, virq#1, P#4*)
6	sched_in_vcpu(*vcpu1, virq#3, P#2*)	27	sched_in_vcpu(*vcpu0, virq#1, P#5*)
7	sched_in_vcpu(*vcpu2, virq#1, P#4*)	28	sched_out_vcpu(*vcpu2, reason = hlt, P#4*)
8	sched_in_vcpu(*vcpu0, virq#1, P#5*)	29	sched_out_vcpu(*vcpu0, reason = hlt, P#5*)
9	sched_out_vcpu(*vcpu2, reason = hlt, P#4*)	30	sched_in_vcpu(*vcpu0, virq#1, P#5*)
10	sched_out_vcpu(*vcpu0, reason = hlt, P#5*)	31	sched_out_vcpu(*vcpu0, reason = hlt, P#5*)
11	sched_ttwu(*vcpu1, P#1*)	32	sched_in_vcpu(*vcpu2, virq#1, P#4*)
12	sched_out_vcpu(*vcpu1, reason = hlt, P#2*)	33	sched_out_vcpu(*vcpu2, reason = hlt, P#4*)
13	sched_in_vcpu(*vcpu2, virq#1, P#4*)	34	sched_in_vcpu(*vcpu0, virq#1, P#5*)
14	sched_in_vcpu(*vcpu0, virq#1, P#1*)	35	sched_out_vcpu(*vcpu0, reason = hlt, P#5*)
15	sched_in_vcpu(*vcpu1, reason = hlt, P#5*)	36	sched_ttwu(*vcpu1, P#1*)
16	sched_out_vcpu(*vcpu1, reason = hlt, P#5*)	37	sched_out_vcpu(*vcpu1, reason = hlt, P#3*)
17	sched_out_vcpu(*vcpu2, reason = hlt, P#4*)	38	sched_in_vcpu(*vcpu2, virq#1, P#4*)
18	sched_in_vcpu(*vcpu1, virq#1, P#5*)	39	sched_out_vcpu(*vcpu2, reason = hlt, P#4*)
19	sched_out_vcpu(*vcpu1, reason = hlt, P#5*)	40	sched_in_vcpu(*vcpu0, virq#1, P#1*)
20	sched_in_vcpu(*vcpu2, virq#1, P#4*)	41	sched_in_vcpu(*vcpu1, virq#1, P#5*)
21	sched_out_vcpu(*vcpu2, reason = hlt, P#4*)	42	sched_out_vcpu(*vcpu1, reason = hlt, P#5*)

Algorithm 10: Process Ranking Algorithm (PRA)

```
if event == sched_ttwu then
    j = getVMvCPU(wakee_tid);
    k = getVMvCPU(waker_tid);
    wakerCR3 = getLastCR3(vCPU_k);
    updateWaker(vCPU_j, wakerCR3);
else if event == vm_inj_virq then
    j = getVMvCPU(tid);
    pCR3 = getVMvCPU(vCPU_j);
    if vec == IPI then
        wakerCR3 = queryWaker(vCPU_j);
        LINK_HORIZONTAL(wakerCR3, pCR3);
for all process cr3_i ∈ CR3 do
    R(cr3_i);
connected_subgraph = breadth-first-search(CR3);
customized_graph = denoise(connected_subgraph, R(CR3));
```

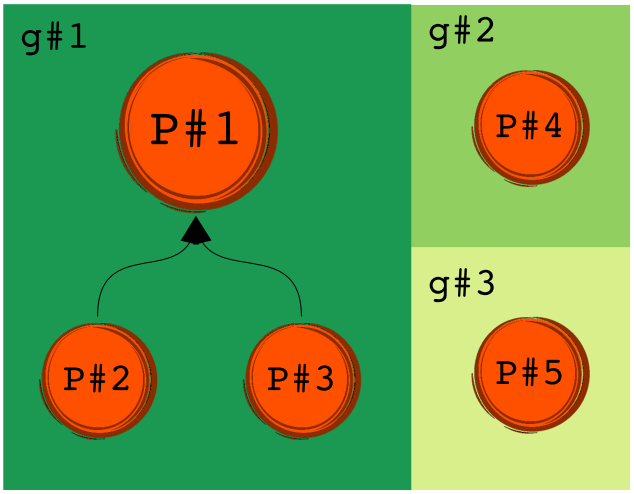


Figure 6.2 Connectivity graphs for different groups of processes. Here, three groups of g#1, g#2, and g#3 are shown.

vCPU execution period which uses `sched_in` and `sched_out`. For simplicity, we grouped a sequence of events into one event in Table 5.1. `sched_in_vCPU(vcpu, vec, cr3)` represents the sequence of `sched_in(vcpu)`, `vm_inj_virq(vec)`, and `vm_entry(cr3)` events. Furthermore, `sched_out_vCPU(vcpu, reason)` shows the sequence of `vm_exit(reason)`, `sched_out(vcpu)` events.

Figure 6.3-a presents vCPU detection algorithm, using a sequence of events shown in Table 6.1. The first event, `sched_in_vCPU` event, shows that the vCPU2 is being scheduled in on pCPU and the vCPU state goes from the blocking to the running state. Then, the second event (`sched_out_vCPU` event), shows that the pCPU is yielded and the state changes to blocking. This algorithm is simplistic and just represents the running and blocking states. It cannot detect the reason of blocking state. WRA is used to detect the reason for blocking.

Figure 6.3-b presents a vCPU view using the events in Table 6.1. The key idea is that the event indicating the cause of the blocking state is unknown a priori. The WRA algorithm uses the virtual injected interrupt to reveal the blocking state. For example, event #6 depicts that the vCPU1 is being scheduled in and the reason for blocking was waiting for Disk.

Figure 6.3-c presents a process view using the WRA algorithm. The WRA algorithm uses the CR3, which is the base pointer to the page table of a process, to identify the running process on the vCPU. For example, event 25 shows that vCPU1 is scheduled in and the reason for blocking was Network. It also depicts that P#3 is the process that was waiting for the packet to receive from the network.

Figure 6.3-d shows the customized vCPU view, using the significant processes detection

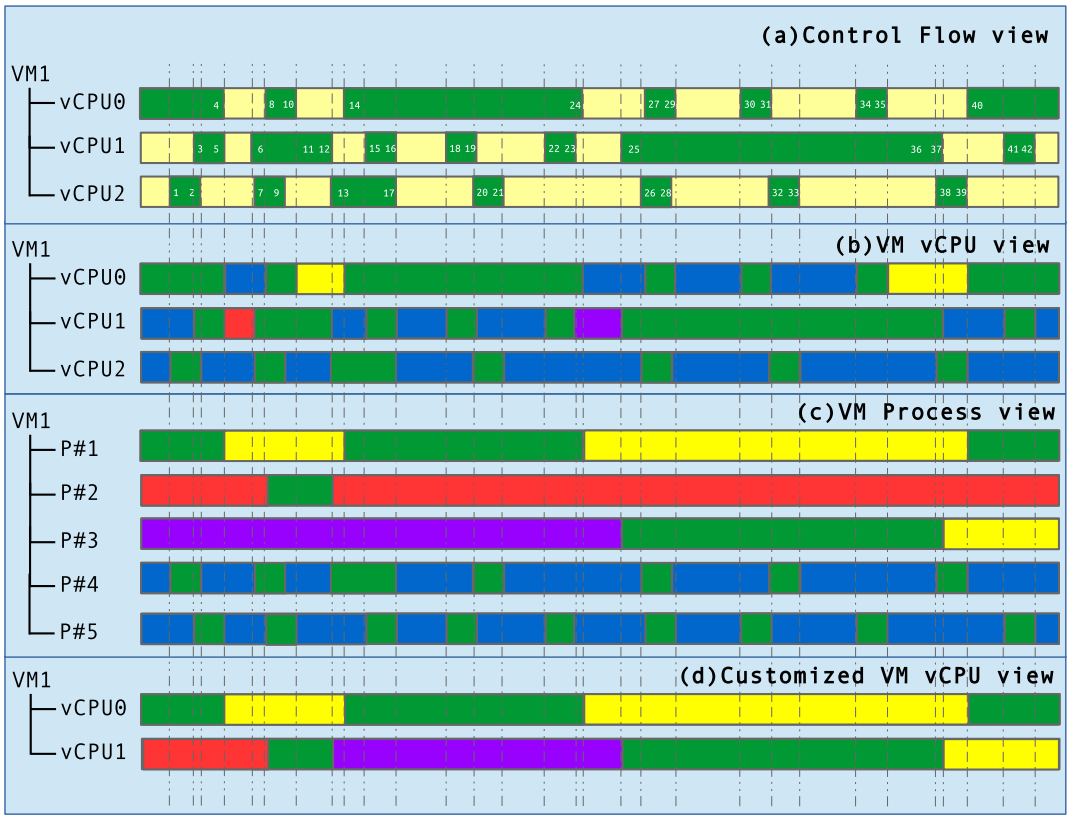


Figure 6.3 Example

algorithm. The PRA algorithm computes the process rank and filters the processes that are not significant. Then, it aggregates their features and builds a new VM feature. For this example, we consider the group that contains the first significant process. In this case, the customized VM vCPU view shows the aggregated states for P#1, P#2, and P#3.

6.4.5 Feature extraction and feature selection

In this section, the metrics and features extracted from the VMs are explained. Here, the events of each VM are monitored by our agent-less tracing mechanism. Since each minute of tracing data could include thousands of events, we need to apply a feature extraction strategy to have a more compact data representation. Thus, based on our domain knowledge, we define the feature extraction approach with several components, as shown in Table 6.4. These features provide a simple representation of the complex original data that could lead to a more suitable input for the learning algorithm. In addition, we apply a feature selection strategy to feed a subset of the initially selected features to the clustering phase. The advantage of feature selection is when the original features are very diverse and might mislead the learning

algorithm.

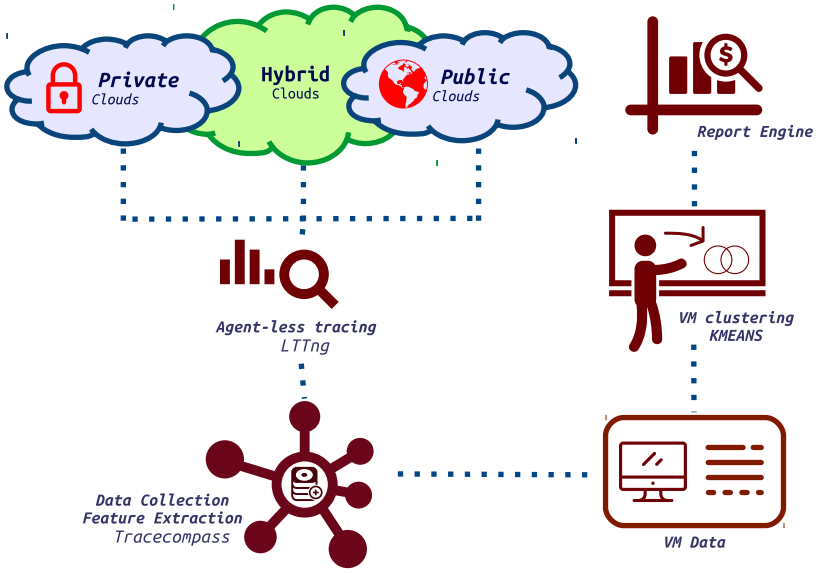


Figure 6.4 Architecture of our implementation

6.4.5.1 Trace data collection

The Linux Trace Toolkit Next Generation (LTTng) [1] is used to monitor the workload on each VM. LTTng is a lightweight and open-source tracing tool on Linux which provides detailed information on the interactions between the kernel and user-space. In addition, we use the Kernel Virtual Machine (KVM) [128] as our hypervisor-based virtualization, which is compatible with LTTng.

In our work, the collected tracing logs are fed to the Tracecompass [2] open source tool, which is designed to view and analyze traces using pre-built modules. Since Tracecompass does not support agent-less VM analysis, we have implemented the WRA algorithm as a module in Tracecompass. Our designed WRA module collects useful information from streaming trace logs, and stores it into a database called State History Tree (SHT). The SHT data structure uses a disk-based format to store large interval data on permanent storage with a logarithmic access time [129]. Figure 6.5 shows the structure of the SHT designed in our analysis. For instance, each VM with its specific ID has different vCPUs, named by their CPU ID. Each vCPU can be in various states, as shown in Figure 6.1. Also, the same information is stored for each process.

Once the WRA algorithm constructs the SHT structure, useful metrics are extracted by querying the SHT attributes. In total, 25 metrics are considered per VM. Table 6.2 provides

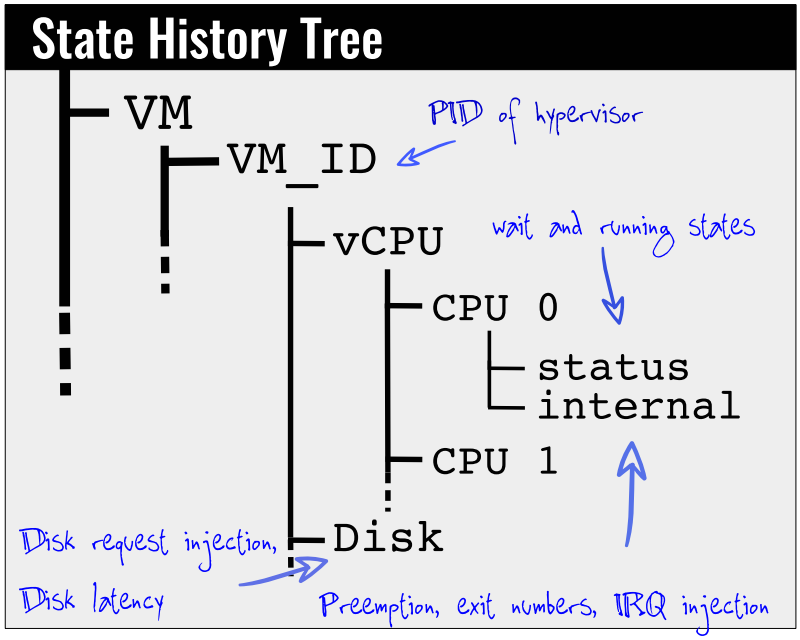


Figure 6.5 State History Tree used to store different information of VM

a complete list of the metrics, along with a short description. The W_X metric, where $X \in \{Disk, Net, Timer, Task\}$, is the average time that a vCPU waits for a signal to be waken up. The E_X with $X \in \{Root, non - Root\}$ metrics are the average time that a VM is running in a VMX root or VMX non-root mode. The f_X with $X \in \{Disk, Net, Timer, Task\}$ metrics show the waking up frequency of a vCPU by different signals. W_X, E_X and f_X are calculated based on the status attribute of each VM in the SHT. The I_X, where $X \in \{Disk, Net, Timer, Task\}$, metrics are the frequency of injecting virtual interrupts into a VM.

6.5 VM Workload Clustering Model

Since no labeled training data is available, we propose to apply the K-Means unsupervised algorithm [130] to cluster VMs into distinct groups. Unlike supervised classification approaches, which require a training dataset to assign a predefined label to a VM, a clustering approach groups VM workloads based on the actual workload distribution. Therefore, it potentially discovers interesting distinct VMs clusters, where each one could represent one specific workload type. For this purpose, each data sample x_i is defined as following:

$$x_i = (f_{i1}, f_{i2}, ..., f_{ij}, ..., f_{iM}) \tag{6.2}$$

Table 6.2 The complete list of metrics extracted from streaming of trace data

Term	Features collected by tracing
W_{Disk}	Wait for Disk average time
W_{Net}	Wait for Net average time
W_{Timer}	Wait for Timer average time
W_{Task}	Wait for Task average time
E_{Root}	Root mode average time
$E_{non-Root}$	Non-Root mode average time
f_{Disk}	The frequency of wait for Disk
f_{Net}	The frequency of wait for Net
f_{Timer}	The frequency of wait for Timer
f_{Task}	The frequency of wait for Task
I_{Disk}	Virtual disk irq injection rate
I_{Net}	Virtual network irq injection rate
I_{Timer}	Virtual timer irq injection rate
I_{Task}	Virtual task irq injection rate
FP_{VMVM}	The frequency of two VMs preempt each other
FP_{HostVM}	The frequency of host preempts a VM
FP_{VMProc}	The frequency of VM processes preempt each other
$FP_{VMThread}$	The frequency of VM threads preempt each other
N_{Exit}	The frequency of different VM exit reason
f_{Read}	The frequency of read from Disk
f_{Write}	The frequency of write to Disk
B_{Read}	Total Block Disk Read
L_{Read}	Total Latency Disk Read
B_{Write}	Total Block Disk Write
L_{Write}	Total Latency Disk Write

where M is the maximum number of workload related metrics that are considered as features in our analysis.To enable meaningful comparisons, sample vectors are individually normalized using the $L2$ norm, such that they become of equal magnitude [131], that is,

$$\sum_{m=1}^{M} f_{im}^2 = 1, \ \forall i = 1..N. \tag{6.3}$$

The K-Means clustering algorithm finds the clusters $C_1, ..., C_K$ over the data matrix $X_{N,M}$, where N is the total number of VMs and M is the number of features. K-Means is a representative-based algorithm in which the clustering is performed based on a group of centroids (partitioning representatives). Centroids are hypothetical sample points placed at the center of each cluster. Given a certain number of centroids, K, a distance function can be used to assign the data points to their closest representatives. In the K-Means algorithm, the sum of squares of the Euclidean distances of data points to their closest centroids is used to quantify the objective function of the clustering as

$$\min_{C} \sum_{j=1}^{K} \sum_{x_i \in c_j} \|x_i - \mu_j\|^2 \tag{6.4}$$

where the set C is $\{c_1, ..., c_k\}$. The term c_j shows the set of points that belong to cluster j and μ_j is the mean of points in c_j.

An advantage of using the K-Means algorithm in our VM analysis approach is that each centroid will represent the workload of its cluster. This provides a condensed useful information about the behavior of many VMs.

Once the VM workload clustering is determined, it is important to evaluate its quality. As we know, clustering validation is often difficult in real datasets, since the problem is defined in an unsupervised way. Here, we use the silhouette score as an unsupervised validation criteria to interpret and evaluate the consistency of the clustering results. The silhouette coefficient measures how similar a sample VM x_i is to its own cluster (cohesion), compared to other clusters (separation). Let in_i be the average distance of x_i to other VM data points within its cluster and out_i be the minimum average distance of x_i to points in other clusters. The silhouette score s_i, specific to the ith sample, is computed as:

$$s_i = \frac{out_i - in_i}{max(in_i, out_i)} \tag{6.5}$$

The silhouette score ranges from -1 to $+1$, where large positive values indicate highly sepa-

rated clustering and negative values represent some level of data points mixing from different clusters. Having in_i as close as possible to zero indicates a very cohesive cluster, and a large out_i value represents a large separation from the closest neighboring cluster. A negative score is undesirable, since it corresponds to the case where the average distance to in-cluster samples exceeds the minimum average distance to out-cluster data points, i.e., $(in_i > out_i)$. It should be noted that, while the quality of a clustering algorithm plays an important role in finding good clusters, having cohesive and well separated clusters also largely depends on the nature of the data and the distance metric. Reporting the average silhouette coefficient over all points of a single cluster indicates how densely the points of the cluster are grouped. The average silhouette score over all data points of the entire dataset represents how appropriately all sample points are clustered.

A two step clustering approach is employed in our analysis, where firstly all VMs are grouped into a set of clusters. Then, the second stage clustering is applied to each discovered cluster in the first stage. Here, each VM would be characterized by a tuple (Ci, Cj), indicating the cluster it belongs to, in the first and second stage respectively. This maintains a better grouping of VM workloads by first performing a coarse grain characterization of VMs, followed by a more detailed clustering. Therefore, we believe that the idea of the two stage clustering provides a good intuitive feel of both coarse and fine grain workload characterization. It should be noted that the number of clusters selection criteria is always guided by the total silhouette score. That is, at any stage we select the clustering with maximum total silhouette score as the desired solution.

6.6 Experimental Evaluation

In this section, we present the experiments on our proposed two stage clustering approach. Different types of workloads are generated using well-known Linux applications (e.g., MySQL, Apache, dd, Sysbench, netSpeed, wget, etc.). The list of applications along with their description is shown in Table 6.3. We broke down these applications into three categories based on their behavior. Then, we mixed them to generate over 60 different workloads and scenarios. Our database is available online at [126]. As mentioned in Section 6.4, once feature metrics are collected, a de-noising PageRank based algorithm can be used to filter out non important processes. Here, we present the results of the VM clustering method with and without the de-noising strategy.

In the first experiment, we use the VM execution time in VMX root and VMX non-root states along with the virtual interrupt injection rate for the whole VM as our feature vector. As we show in the following section, the injection interrupt rate could expose the behavior

Table 6.3 List of applications for workload generation

Application Behavior	Application Name	Application Description
CPU	Sysbench, Stress, Burn, Chess, Compress, encode, interBench, openSSL, smallpt, infinitLoop	Bunch of well-konwn benchmarking tools for Linux to stress CPU including computing prime number, compressing files, video and audio encoding, mathematic operations
Disk	Sysbench, Compress, dbench, dd, interBench, hdparam, fsMarker, aio, ioZone, pgbench-disk, pqlight, stream, tioBench	Include different testing toolkits for Disk I/O. These toolkits provide realistic scenarios for reading/writing with different options to evaluate Disk I/O.
Net	Wget, iperf, netperf, netping, netSpeedTest, scp, ab, httperf	Include real network applications in Linux. It covers web server, file transfer server, downloading file.

of VMs since it shows the communication between processes and host devices. Note that our feature vectors are normalized, and their absolute values are of no significance.

At the first stage of the clustering, we realize that applying K-Means to detect three coarse clusters yields the best total silhouette score of 0.42, and per cluster silhouette scores of $C0 = 0.38, C1 = 0.35, C2 = 0.48$. Interestingly, this matches the reality since our generated workloads are divided into three types (CPU, Disk, and Network).

Figure 6.6 depicts the centroids of the first stage clustering. The prototype usage pattern in Figure 6.6 shows a high CPU usage and high timer interrupt rate for the first cluster (C0), which could represent a CPU intensive cluster. As mentioned before, the VMM executes the privileged instructions in a safe environment, VMX root mode. Then the VM resumes to VMX non-root mode after handling the privileged instruction. In Linux, the KVM module clears the CPU preemption flag before going to the VMX non-root mode. As a result, no one can preempt the VM in VMX non-root mode. In order be able to yield the pCPU, the Linux scheduler kicks the VM by injecting external interrupts into the vCPU. This causes an exit to VMX root mode (i.e., exit number one) in order to handle the exit reason. In this state, the preemption flag is set in order to authorize the host scheduler to preempt the VM. As a result, the CPU intensive VMs have a high rate of timer interrupt, along with high CPU usage in VMX non-root mode.

The second cluster (C1) represents the Disk intensive VMs. The prototype usage pattern of Disk intensive VMs includes a large disk interrupt rate along with a high task interrupt rate. This is an indication of disk intensive VMs, waiting more frequently for some task to

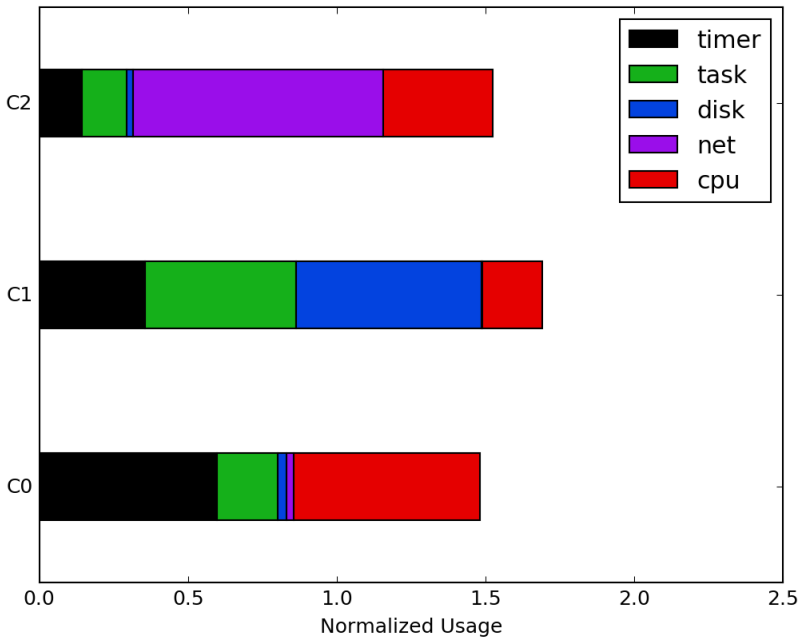


Figure 6.6 Characteristic of workloads discovered in first stage of clustering

finish I/O. The third cluster (C2) is found to include Network intensive VMs. The network intensive VMs have high network interrupt rates. The processes in network intensive VMs wait more for processes on other machine, compared to disk intensive ones which wait more for processes in the same machine.

The result of the first stage can be used by the cloud infrastructure administrator to optimize resource usage. A host with too many VMs from same cluster could encounter resource contention. For instance, assigning too many CPU intensive VMs to one host machine causes preemption on physical CPUs, which leads to latency. Thus, using our clustering method, the administrator can balance the host resource utilization and reduce resource contention.

In the second stage, we apply K-Means to each of the first stage clusters independently, and choose the clustering with the largest silhouette score. For our particular set of workloads, the first stage cluster C0 turns out to best characterized as five distinct groups, with a total silhouette score of 0.59. Figure 6.7 shows the similarity plot for the five clusters found in the second stage. Each point on both the X and Y axis corresponds to a particular VM, which is labeled once on the Y axis. The second stage cluster index corresponding to this VM is enclosed in brackets.

Each point on the similarity plot presents the normalized Euclidean distance, defined as in (6.6), between VM feature vectors intersecting at that point. Note that d_{ij} is the actual

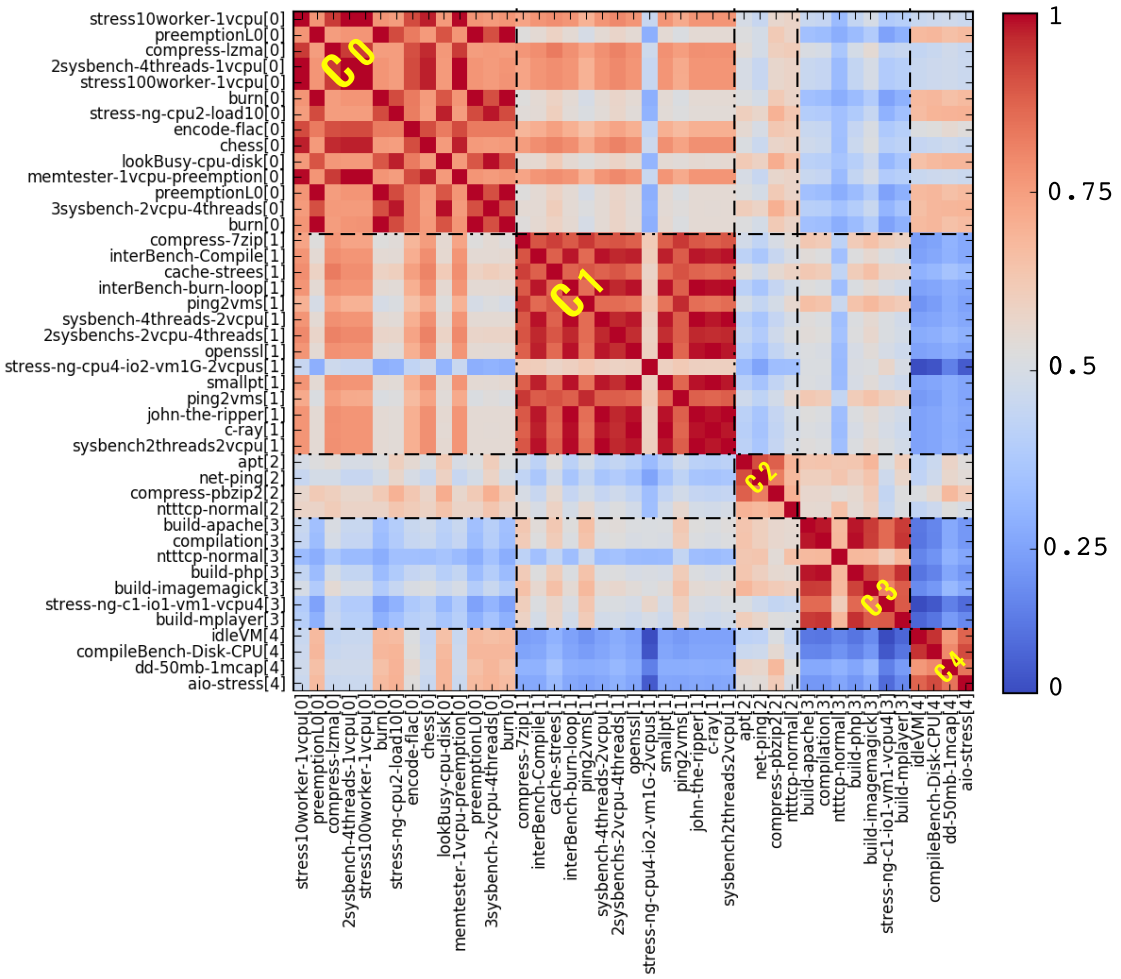


Figure 6.7 Similarity plot for the five workload clusters discovered in the first group, i.e., (C0, Cj), $j \in [0, 4]$ (Silhouette = 0.59).

Euclidean distance between VM i and j, while d_{max} and d_{min} correspond to the maximum and minimum distances, respectively. The similarity score sim between any two VMs ranges from zero to one. Interestingly, sorting VMs according to their cluster index results in a similarity plot that resembles a block diagonal shape, for cohesive and well separated clusters.

$$sim_{ij} = 1 - \frac{d_{ij} - d_{min}}{d_{max} - d_{min}} \tag{6.6}$$

Figure 6.7 illustrates such a block diagonal pattern, revealing how good the clustering is. This is also reflected in Figure 6.8a as the computed silhouette scores for each cluster show positive and mostly large values. Figure 6.9a depicts the characteristics of workloads in five different types of workloads. These groups mainly differ in the amount of CPU usage as

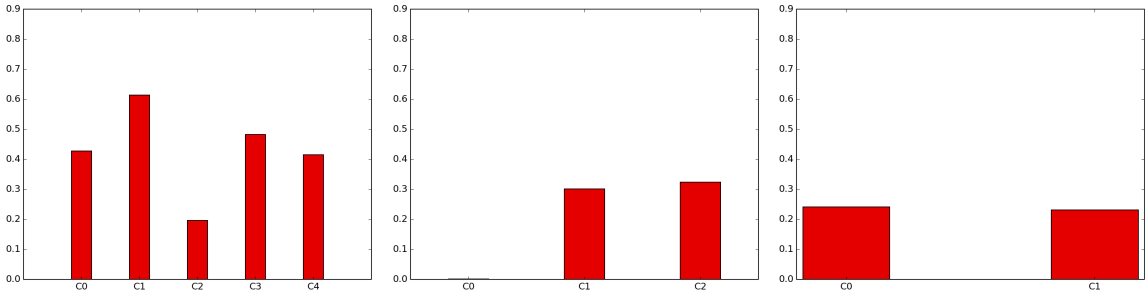
well as timer and task interrupts. For example, there are two groups (C0, C2) and (C0, C3) which have high timer interrupts, reflecting multiple processes dependencies. Interestingly, (C0, C4) is almost entirely timer intensive, with minimal CPU usage, indicating that the VMs in this group represent mostly idle workloads. Instances in this group do not cause any contention and could be deployed in any machine.

Let us elaborate on a few interesting observations from the similarity plot for C0. The first interesting observation is regarding VMs in clusters (C0, C3) which have a high task interrupt rate. The processes inside these VMs are either dependent on other processes or dependent on their own threads. Interestingly, all VMs that build an application are in this group, since making a file depends on compiling other files. Figure 6.10 shows an example of this group. As shown, processes wait for other processes to finish their job.

Moreover, there is a moderate similarity among VMs in clusters (C0, C1) and (C0, C0). This is visible from the moderately warm areas at the intersection of these two clusters. Further inspection of the VMs reveals that these two groups are both CPU intensive, with negligible task dependency. It means that both group have almost single threaded processes. This is clearly observed from the centroids for C1 and C0 in Figure 6.9a, further showing a sharp contrast in CPU and timer composition among the two clusters. VMs in cluster (C0, C0) have a special behaviour. Figure 6.11 depicts one instance of this cluster. As shown, it executes a small amount of code and then waits for a timer to be fired. Other VMs could be affected by this catastrophic behavior. Some instances of this group do not utilize the pCPU and preempt other vCPUs running on the pCPU. Thus, all instances with preemption at the host level are in this group. In contrast, VMs in cluster (C0, C1) are highly CPU intensive and utilize the pCPU all the time. Despite this overlap, Figure 6.8a shows that the silhouette score for (C0, C1) is significantly larger than (C0, C0), suggesting that C1 is a better cluster. This is because (C0, C1) is a more cohesive cluster compared with (C0, C0), as suggested by the very warm texture of (C0, C1). This implies that the VMs in this group are very similar and closely resemble the prototype centroid representing this cluster. This information can be useful for the cloud administrator when dealing with deployment of these very similar workloads. The VM deployment strategy should satisfy both IaaS providers and VM users. It should guarantee an efficient usage of host resources, while avoiding resource contention between VMs.

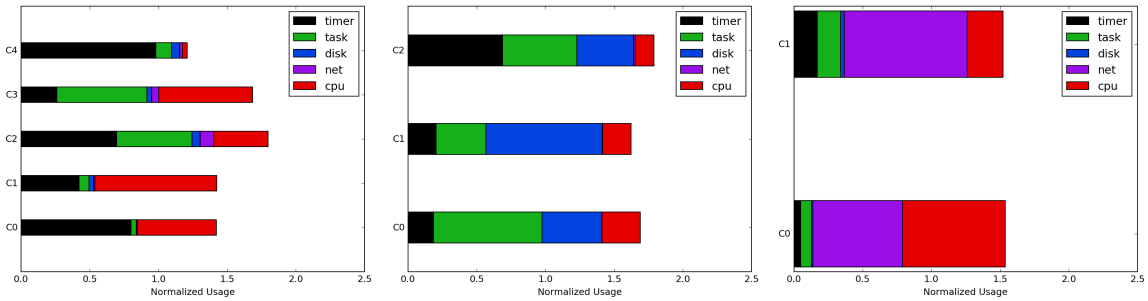
Another interesting observation is that two particular VMs in (C0, C3) and (C0, C1) are quite dissimilar to their co-cluster VMs. These are the `ntttcp-normal` workloads, which moderately use the network but are put into these clusters due to making significant use of timer and task as well as CPU resources. While it is debatable to put these workloads in the

same group with non-network VMs, the similarity plot can easily visualize such peculiarities for the system administrator.



(a) (C0, Cj), $j \in [0, 4]$ (b) (C1, Cj), $j \in [0, 2]$ (c) (C2, Cj), $j \in [0, 1]$

Figure 6.8 Silhouette score of VM clusters discovered in second stage of clustering. Cj represents the second stage cluster index.



(a) (C0, Cj), $j \in [0, 4]$ (b) (C1, Cj), $j \in [0, 2]$ (c) (C2, Cj), $j \in [0, 1]$

Figure 6.9 Characteristic of workloads discovered in the second stage of clustering. Cj represents the second stage cluster index.

The second and third stage one clusters are also further processed into three and two groups of workloads by our approach. Figures 6.12 and 6.13 provide similarity plots for C1 and C2, respectively. As for (C0, C3) and (C0, C1), (C2, C1) also includes a rather dissimilar VM, namely `host-slow-disk`, which, although highly network intensive, also makes moderate disk usage. The `host-slow-disk` VM executes `scp` to copy a big file from one VM to the host. For this special VM, we put a cap on disk usage to slow down the data reading. This is realistic, since many IaaS providers like Amazon limit number of Block I/O accesses in order to reduce disk contention. As a result, we could call the VM more network intensive rather than disk intensive. Therefore, it lies somewhere between the disk intensive and network intensive groups, but closer to the latter one.

Another peculiar case is (C1, C0) with a cluster silhouette score of almost zero, as indicated

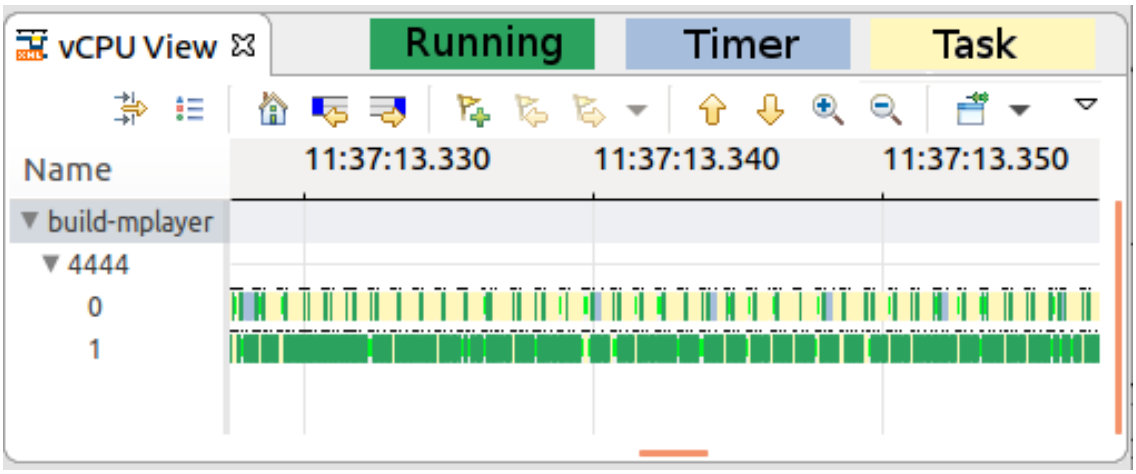


Figure 6.10 vCPU view for the build-mplayer VM that has a high task dependency and high CPU utilization

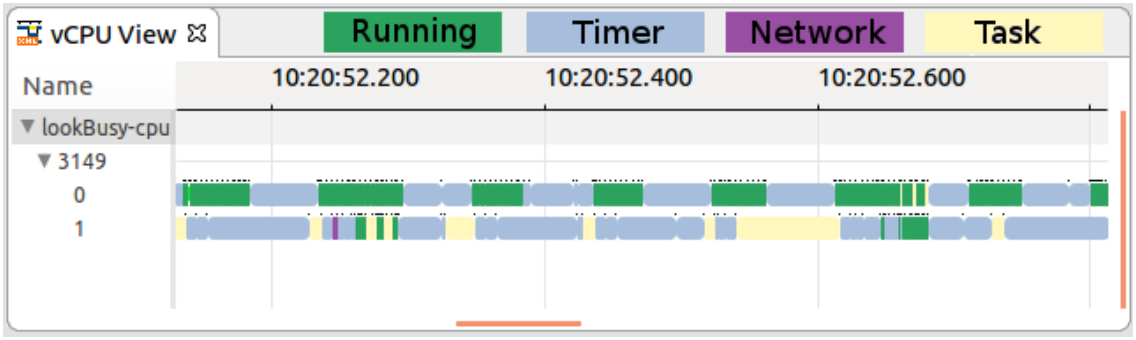


Figure 6.11 One of the instances from cluster (C0, C0)

in Figure 6.8b. Inspecting the similarity plot of Figure 6.12 for C1 reveals a rather loose cluster in this case (i.e., C0). While Figure 6.9b shows that the centroid for this cluster (C0) is highly task dependent, further inspection of the feature vectors extracted for the VMs in (C1, C0) reveals that they are rather diversely affected by timer interrupt and CPU usage. Such cases, with low cluster quality, are generally not of interest from a resource management point of view. However, they can still be used to investigate the root cause of performance issues through their representative prototype workload.

VMs in cluster (C1, C2) read or write a large number of files at once and then wait for some time and then continue. In contrast, VMs in cluster (C1, C0) read small data chunks. Reading small files is extremely harmful for other VMs since it issues a number of disk requests to the host level and cause contention.

Let us now elaborate on the workload prototypes discovered in the second stage, which are

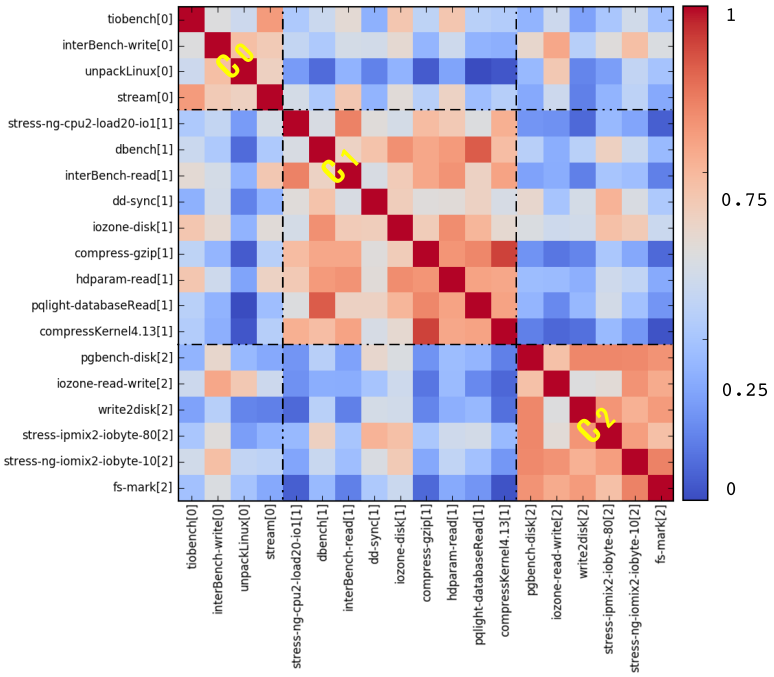


Figure 6.12 Similarity plot for the three workload clusters discovered in the second group, i.e., (C1, Cj), $j \in [0, 2]$ (Silhouette = 0.40).

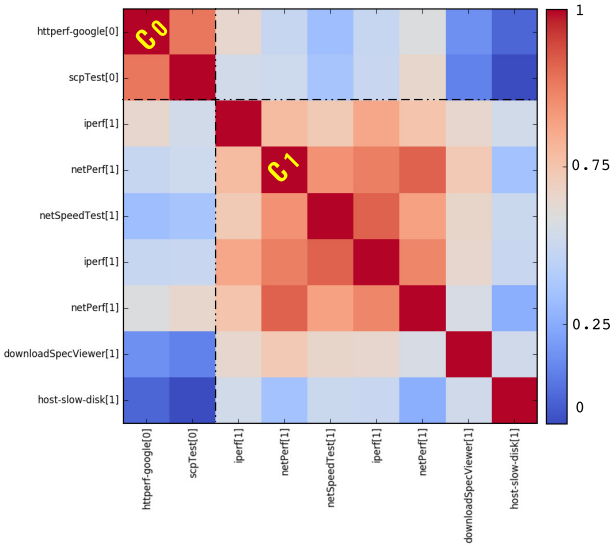


Figure 6.13 Similarity plot for the two workload clusters discovered in the third group, i.e., (C2, Cj), $j \in [0, 1]$ (Silhouette = 0.46).

shown in Figures 6.9a, 6.9b, and 6.9c, corresponding to first stage clusters C0, C1, and C2, respectively. Recall that C0 included CPU intensive VMs, while C1 and C2 were discovered to be disk and network intensive, respectively. We start from the network intensive group C2, which was found to have only two clusters of workloads; (C2, C1) which is highly network intensive and (C2, C0) with also significant CPU usage. The latter group should be allocated adequate CPU resources to prevent bottlenecks, whereas the former group can be co-located with CPU intensive VMs on a server which has otherwise no network usage. Hence, while VMs in (C2, C0) are compatible with those of all other first stage groups, (C2, C1) may only be co-located on physical servers hosting (C1, C2) or (C0, C4).

As for the disk intensive group, we have discovered three clusters with contrasting behaviors in the composition of disk and task usage as well as timer interrupt rates. More specifically, we observe two contrasting sets of workloads (C1, C1) with large disk usage but small inter-task dependencies and (C1, C0) with moderate disk usage and significant task dependencies. Alternatively, (C0, C0) and (C0, C1) make heavy use of CPU and timer resources. This indicates that these VMs contain contending processes which are CPU intensive.

In the next experiment, we applied the de-noising PRA algorithm on our data to find significant processes in each VM. Then, we built a new feature vector based on significant processes. Similar to previous experiments we followed a two stage clustering strategy.

Figure 6.14 illustrates the similarity plot for the C0 cluster which is CPU intensive. The first stage cluster C0 turns out to be best characterized by five distinct groups, with a total silhouette score of 0.64. According to Figure 6.14, there is low inter cluster similarity, which is clear from the moderately cold areas at the clusters intersections.

There is almost no similarity among VMs in clusters (C0, C1) and (C0, C0). Further investigation of the VMs reveals in clusters C1 and C0 that these two groups are both CPU intensive. However, C0 VMs are executing more task dependent processes, while VMs in C1 have high timer interrupt rate.

Moreover, there is a moderate similarity among VMs in clusters (C0, C1) and (C0, C3). This is visible from the moderately warm areas at the intersection of these two clusters in the similarity plot of Figure 6.14. Further inspection of the VMs reveals that these two groups are both CPU intensive, with high timer dependency. This means that both groups have almost single threaded processes. This is clearly observed from the centroids of C1 and C3 in Figure 6.15, further showing a sharp contrast between CPU and timer workloads among the two clusters.

Another special cluster is (C0, C2), with a cluster silhouette score of almost zero, as indicated

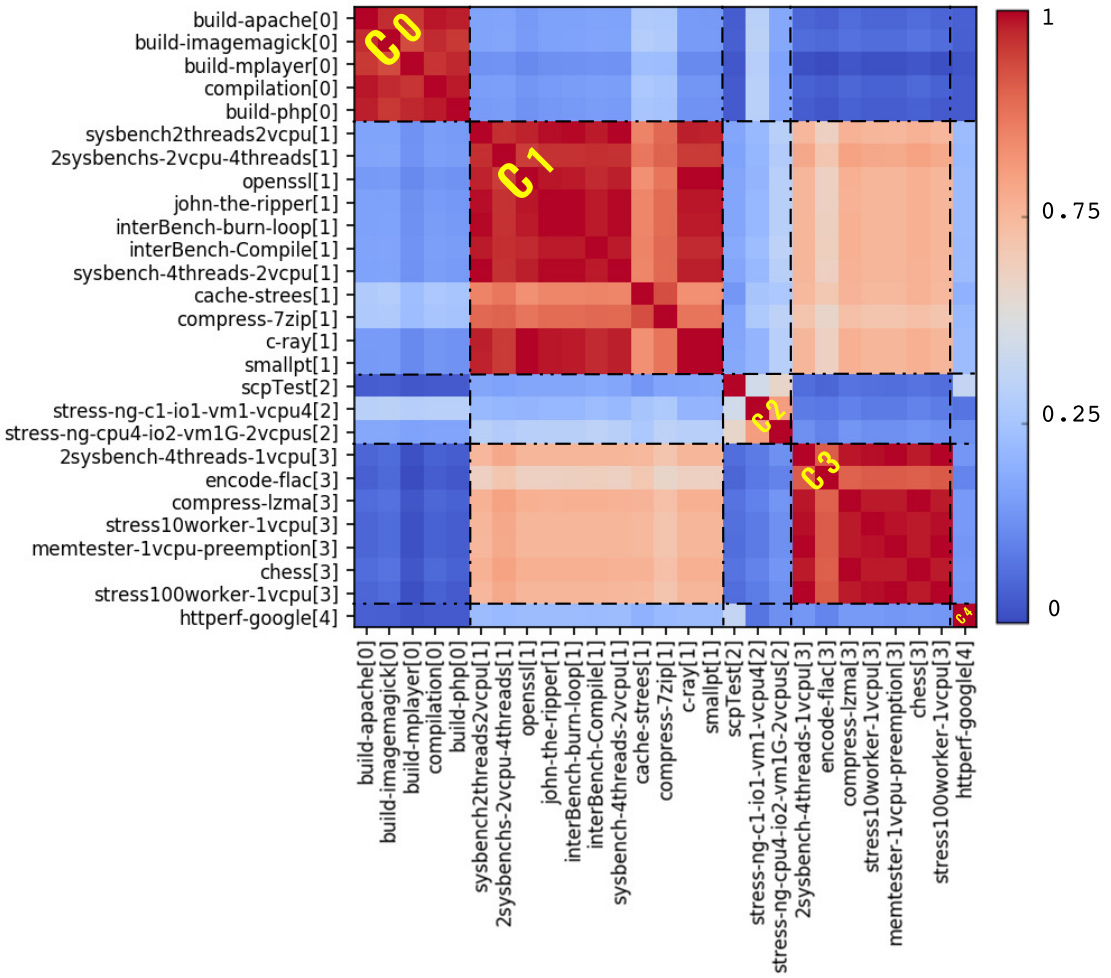


Figure 6.14 Similarity plot for the five workload clusters discovered in the first group using PRA algorithm, i.e., (C0, Cj), $j \in [0, 4]$ (Silhouette = 0.64)

in Figure 6.16. The similarity plot of Figure 6.14 for C2 reveals a rather loose cluster in this case (i.e., C0). While Figure 6.15 shows that the centroid for the cluster C2 is highly CPU dependent, further inspection of the feature vectors extracted for the VMs in (C0, C2) reveals that they are rather diversely affected by Disk interrupt, Network interrupt and CPU usage. As a result, we could call this cluster a compound cluster (CPU, Disk, Network). Such cases with low cluster quality, but diversity in interrupt rates, are generally interesting from a resource management point of view, as they represent how resources are utilized.

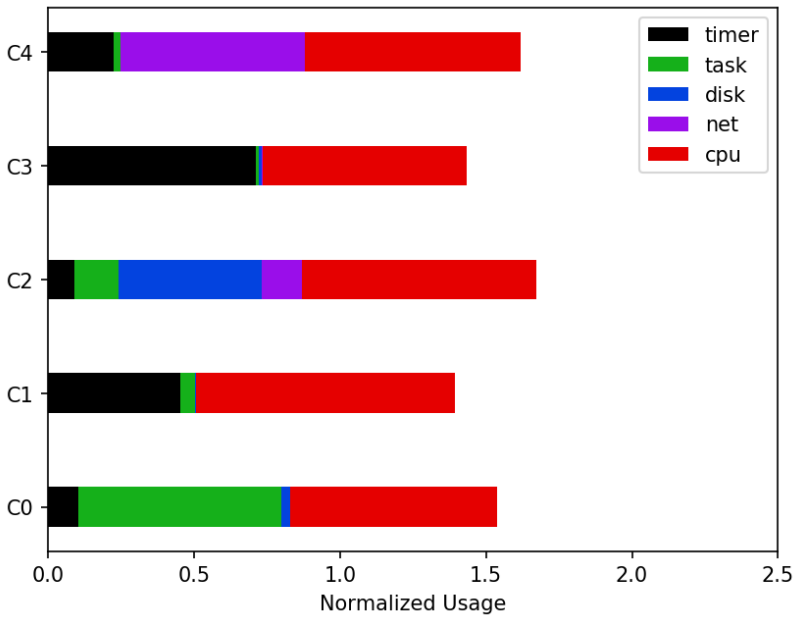


Figure 6.15 Workload characteristic of C0 cluster discovered in the second stage of clustering based on the PRA algorithm.

6.7 Overhead Analysis

In this section, the overhead of the WRA algorithm is compared with the agent-based feature extraction (AFE) method. In order to compare the two approaches, we enabled the tracepoints that were needed for extracting the same features from inside of VM. Also, it is worth mentioning that our WRA algorithm needs only to trace the host. As shown in Table 6.4, the FAE approach adds more overhead in all tests, since it needs to trace the VMs and the overhead of virtualization will be added. We used the Sysbench benchmarks to reveal the overhead of both approaches, since Sysbench is configured for Memory, Disk I/O and CPU intensive evaluations. Our approach has negligible overhead for CPU and Memory intensive tasks at 0.3% or less.

Table 6.4 Overhead analysis of WRA algorithm comparing with agent-based feature extraction method

Benchmark	Baseline	FAE	WRA	Overhead	
				FAE	WRA
File I/O (ms)	450.92	480.38	451.08	6.13%	0.03%
Memory (ms)	612.27	615.23	614.66	4.81%	0.01%
CPU (ms)	324.92	337.26	325.91	3.65%	0.30%

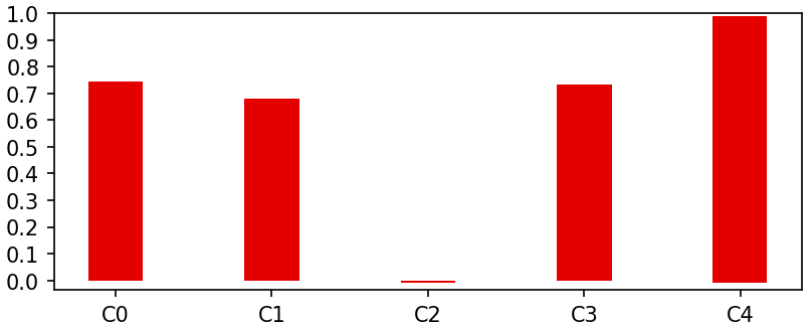


Figure 6.16 Silhouette score of the C0 cluster discovered in the second stage of clustering using the PRA algorithm.

6.8 Ease of Deployment

Our proposed workload analysis technique needs to enable a few host hypervisor tracepoints (five events). However, other available trace-based approaches [73] [74] require to enable most Linux tracepoints in the host and the VMs. The main challenge in these methods is the absence of a global clock, which synchronizes the events collected from the host and the VMs. Therefore, with the cost of an extra overhead additional events are defined as a synchronization strategy. This leads to an increase in the total analysis completion time. On the other hand, our proposed approach receives all events from the same clock source in the host, thus does not require a synchronization technique. In addition, our method can analyze the VMs with different operating systems (e.g., Linux, Windows, and Mac OS), since it only needs to enable a few events in the host.

6.9 Limitations

Our proposed approach gathers information from the host. Thus, detailed information regarding the guest operating system is not considered in our method. For instance, our method cannot detect information about containers in a VM, but it can show a container as a separate process. However, the other trace-based methods [73] and [74] provide more information about the processes and their interactions with the guest kernel.

6.10 Conclusions

Efficient distributed resource management, in cloud environments with thousands of virtual machines (VMs), is a demanding task which requires detailed information regarding the behavior of each VM. In this paper, a host level hypervisor tracing approach was proposed,

based on an agent-less feature extraction technique that provided fine grain characterization of VM behavior. A PageRank algorithm was developed to select features from significant processes running on the VMs. A two phase K-Means clustering approach was applied to group VMs which had similarity in terms of workload behavior. The first phase clustering provided a general vision on the behavior of each VM, while the second phase enabled us to discover detailed information about the root causes of different issues. Experiments on a real dataset, including various VMs running different software applications, showed that our proposed method could model VMs behavior as expected. According to our benchmarks, our proposed method reached an accumulated overhead of around 0.3%, whereas other approaches showed an accumulated overhead in a range of 3.65% to 6.13%. Future work includes incorporating other existing tracing based metrics, as additional features in the VM analysis, to reveal more detailed information on the root causes of the VM behavior.

CHAPTER 7 GENERAL DISCUSSION

In this chapter, we observe the broad impact of our work in our domain and in industry, outlining specific contributions, outcomes and limitations of our research.

7.1 Revisiting Milestones

In this section, we further discuss the points introduced in Chapter 3.

Nested VM analysis A part of our work, presented in the first article, dealt with the identification of nested VMs while there is no access inside the VM. We proposed an approach that looked at the exit reason, in the transition from VMX non-root to root mode, to find the guest hypervisor CR3. In the case where the exit reason is `resume` or `launch`, the last visited CR3 is marked as the guest hypervisor CR3. Moreover, the upcoming CR3 is marked as the nested VM process CR3. We developed a graphical view for nested VMs showing the vCPU threads of the nested VMs with their code execution level and states. Then, we experimented with popular applications (e.g., Hadoop) to study the behavior of VMs regarding the overcommitment of resources and other possible problems.

VM vCPU and Process State Detection For the second milestone, we focused on proposing a fine-grained VM vCPU and processes state analysis (including reasons for blocking) based on host tracing. Our method can detect all the running and waiting states for an arbitrary nesting depth. We implemented a graphical view for processes and VMs vCPUs. Our graphical view presents a timeline for each process and vCPU, with different states along with their interactions with the VMM. We demonstrated the power of the proposed technique through different representative use cases based on well-known applications like Apache, MySQL, Hadoop.

Critical Path Analysis Owing to their ability to quickly identify performance sensitive paths in the modern distributed computing landscape, we set our third research milestone as finding the critical path of processes inside VMs from host tracing. It helps the IaaS administrator to construct the process dependency graph across a distributed environment with nested virtualization layers. It is able to find the execution pattern of an arbitrary process inside a VM. It unveils the process resources usage along with its interactions with other contending processes. We then experimented with real software applications, like the Linux Advanced Packaging Tool (APT) and IP Multimedia Subsystem (IMS), and other use cases inspired from industry.

Virtual Machine Workload Clustering While working on the VM state analysis and critical path analysis milestones, we observed that we could extract some of our analysis output as a feature vector, to automatically cluster VMs based on their workload. We employed various feature selection strategies and assessed the quality of the resulting workload clustering. We adopted a two stage feature selection approach, in addition to a one shot clustering scheme. To validate our work, we built a database of actual software applications. The experimental results showed that our method could group VMs with similarity. This information can be used by 1) data center administrators to gain deep visibility into the nature of various VMs running on their infrastructure, 2) performance engineers to assist in the root cause analysis of VM issues and 3) IaaS providers to help in resource management based on VM behavior.

7.2 Research Impact

Cloud computing is pervasively used in industry, as a result of the Pay as Use model that it brings. However, current monitoring tools do not provide enough information for troubleshooting and debugging VMs. Moreover, in most cases, access to the VMs is restricted, so administrative tools on the host cannot easily provide information about the state of the VMs. Therefore, there is a need for low-overhead tools that can extract meaningful information from VMs, without internal access.

In this research project, we proposed several techniques for analysing the behaviour of VMs. Our tool analyses the vCPU threads, block I/O threads, and Network threads inside the host and examines the state of the vCPUs, block I/O, and network. Then, we collect some features from VMs and use them to group similar VMs.

Nested virtualization is frequently used in industry for software scaling, compatibility, and security. However, in the nested virtualization context, the current monitoring and analysis tools do not provide enough information about VMs for effective debugging and troubleshooting. We addressed the issue of efficiently analyzing the behavior of such VMs. Our technique can detect different problems, along with their root causes, in nested VMs and their corresponding VMs. Furthermore, our approach can uncover different levels of code execution among all the host and nested VMs layers.

There is a trade-off between resource utilization and overcommitment in the cloud environment. Using our analysis, the cloud administrator can find any contention between VMs and solve the issue by adding more resources or migrating the VMs.

We argue that we can reduce the complexly of analyzing and monitoring VMs by leveraging

the VM clustering technique. Using this technique, VMs that behave almost the same will be grouped, and the amount of monitoring data will be reduced. As a result, the administrator should deal with ten VMs instead of thousands of VMs. As a very simple example, consider the scenario where one system resource is being fully utilized. The clustering technique could point to the group of VMs that suffer more from contention on the specific resource. Then the resource management could be easier by adding more resource or migrating VMs to another host. Detailed information about behavior of VMs could not be revealed only by considering resource usage, it needs a more sophisticated method for feature extraction.

7.3 Limitations

All our approaches limit their data collection to the host, and guest OS specific information is not accessible with our method. For example, the name and PID of processes are OS specific and our method cannot separate processes based on their name. Furthermore, our technique cannot detect a container in a VM, but it could show it as a separate process, using the GTA algorithm. By contrast, other trace-based methods ([87] and [73]) provide more useful insights about running processes and their interaction with the guest kernel.

CHAPTER 8 CONCLUSION

In last the few years, cloud computing became an emerging technology that allows users to access a pool of resources from anywhere and anytime with ease. Many enterprises are beginning to adopt VMs due to the fact that they can always access the latest applications while having no infrastructure to maintain.

From the IaaS provider point of view, the process of debugging and troubleshooting such infrastructure, with hundreds of hosts and thousands of VMs, is extremely complex. Indeed, one issue in a VM can lead to a performance degradation in other VMs. Moreover, monitoring such an infrastructure comes with scalibility issues due to the amount of data that should be analyzed. This complexity could be reduced by automatically finding the group of VMs that could be the cause for an issue or be affected by the issue. As a result, the cloud administrator can quickly focus on a few VMs instead of thousands of VMs. Therefore, clustering VMs can help finding issues, but also to assist the resource management process.

In our research work, the initial goal was to propose a more fine grained as well as precise VM analysis approach, with the help of tracing. We discussed in details how we developed multiple algorithms which can provide useful information about VM processes and help in analysing the behavior of VMs with out accessing the VMs.

In the first part of the work, we developed a novel host-based process and vCPU state detection algorithm, that can not only find the state of running processes but also recover the reason for being idle. Note that our algorithm can detect the state for VM vCPUs and VM processes for any level of virtualization.

In the second part of the work, we proposed and implemented a method to analyse the execution of a distributed and hierarchical virtualzed environment, using scheduling, network and virtual interrupt events. The algorithm developed can answer the typical question of where lies the bottleneck.

In the third part of the work, we introduced a host based VM feature extraction method to extract meaningful information and provide fine grain characterization of VM behaviour. We used the K-means clustering technique to group VMs which have similarity in term of workload behavior. The two phase clustering technique let us apply more directive features in the second phase, in order to more precisely find the root cause of an issue. To validate our work, we built a database of real software applications. The experimental results showed that our method could identify clusters of similar VMs.

Our benchmarks show that the overhead for our approaches, since they just trace the host, is around 0.3%. By contrast, the overhead of other approaches ranged from 3.65% to 6.13%. Moreover, our approach provides more complete information, since it has access to host hypervisor level information.

Based on our research experience, we recommend that future researchers focus on enhancing our feature selection mechanism to use other existing features for further VM analysis. Our database also could be enhanced to include different anomalies and contention scenarios.

BIBLIOGRAPHY

[1] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium) 2006*, 2006, pp. 209–224.

[2] "Trace compass," https://projects.eclipse.org/projects/tools.tracecompass, 2018, accessed: 2018-12-12.

[3] "Lttng documentation," https://lttng.org/docs/v2.10/, accessed: 2019-01-28.

[4] "Implementing openstate with ebpf," https://qmonnet.github.io/whirl-offload/2017/02/11/implementing-openstate-with-ebpf/, accessed: 2019-01-28.

[5] L. L. Silva, K. R. Paixao, S. d. Amo, and M. d. A. Maia, "Software evolution aided by execution trace alignment," in *2010 Brazilian Symposium on Software Engineering*, Sep. 2010, pp. 158–167.

[6] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "Dksm: Subverting virtual machine introspection for fun and profit," in *2010 29th IEEE symposium on reliable distributed systems*. IEEE, 2010, pp. 82–91.

[7] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011.

[8] "10 most powerful iaas companies," http://www.networkworld.com/supp/2012/enterprise2/040912-ecs-iaas-companies-257611.html.

[9] "10 most powerful paas companies," http://www.networkworld.com/slideshow/32927/10-most-powerful-paas-companies.html.

[10] "10 saas delivery companies to watch," https://www.networkworld.com/article/2285750/saas/10-saas-delivery-companies-to-watch.html.

[11] K. H. Masiyev, I. Qasymov, V. Bakhishova, and M. Bahri, "Cloud computing for business," in *2012 6th International Conference on Application of Information and Communication Technologies (AICT)*, Oct 2012, pp. 1–4.

[12] X. Lei, X. Zhe, M. Shaowu, and T. Xiongyan, "Cloud computing and services platform construction of telecom operator," in *Broadband Network & Multimedia Technology, 2009. IC-BNMT'09. 2nd IEEE International Conference on*. IEEE, 2009, pp. 864–867.

[13] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2008.

[14] B. Bitner and S. Greenlee, "Vm – a brief review of its 40 year history„" http://www.vm.ibm.com/vm40hist.pdf?

[15] Oracle, "Introduction to virtualization," http://docs.oracle.com/cd/E20065_01/doc.30/e18549/intro.htm.

[16] L. Y. Xiao, Z. Wang, R. Wang, and H. N. Wang, "Architecture and key technologies of cloud computing," in *Advanced Materials Research*, vol. 756. Trans Tech Publ, 2013, pp. 1953–1956.

[17] I. Corporation, "Intel 64 and ia-32 architectures software developer's manual," in *Intel Corporation*, 2015, pp. 1–3883.

[18] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, 2010.

[19] M. Beham, M. Vlad, and H. P. Reiser, "Intrusion detection and honeypots in nested virtualization environments," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–6.

[20] "Ravello systems: Virtual labs using nested virtualization," https://www.ravellosystems.com/, accessed: 2018-11-16.

[21] A. Fishman, M. Rapoport, E. Budilovsky, I. Eidus *et al.*, "Hvx: Virtualizing the cloud." in *HotCloud*, 2013.

[22] Intel, "s," in *McAfee Deep Defender Technical Evaluation and Best Practices Guide*. 2821 Mission College Boulevard Santa Clara, CA 95054: Intel Press, Aug 2012, pp. 1–48.

[23] "Windows xp mode," https://www.microsoft.com/en-ca/download/details.aspx?id=8002, 2009, accessed: 2018-12-12.

[24] T. Theakanath, "Proactive monitoring," https://devops.com/proactive-monitoring/.

[25] Optimusinfo, "Open-source vs. proprietary software pros and cons," http://www.optimusinfo.com/downloads/white-paper/open-source-vs-proprietary-software-pros-and-cons.pdf.

[26] Y. Han, "On the clouds: A new way of computing," *Information Technology and Libraries*, vol. 29, no. 2, pp. 87–92, 06 2010.

[27] A. Nisar, W. Iqbal, F. Bokhari, and F. Bukhari, "Hybrid auto-scaling of multi-tier web applications: A case of using amazon public cloud," in *Researchgate*, 01 2015.

[28] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 51, pp. 02 110–1301, 2002.

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[30] M. Bunnell, "Dynamic instrumentation event trace system and methods," May 16 2006, uS Patent 7,047,521.

[31] S. Sharma and M. Dagenais, "Hardware-assisted instruction profiling and latency detection," *The Journal of Engineering*, vol. 1, no. 1, 2016.

[32] "Oprofile," http://oprofile.sourceforge.net.

[33] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page.

[34] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, École Polytechnique de Montréal, 2009.

[35] A. Spear, M. Levy, and M. Desnoyers, "Using tracing to solve the multicore system debug problem," *Computer*, vol. 45, no. 12, pp. 60–64, 2012.

[36] A. Martin and V. Marangozova-Martin, "Automatic benchmark profiling through advanced trace analysis," in *European Conference on Parallel Processing*. Springer, 2016, pp. 63–74.

[37] J. Desfossez and M. DESNOYERS, "Lttng-ust vs systemtap userspace tracing benchmarks," 2016.

[38] F. C. Eigler and R. Hat, "Problem solving with systemtap," in *Proc. of the Ottawa Linux Symposium*, 2006, pp. 261–268.

[39] R. McDougall, J. Mauro, and B. Gregg, *Solaris performance and tools: DTrace and MDB techniques for Solaris 10 and OpenSolaris*. Prentice Hall,, 2006.

[40] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 26:1–26:33, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3158644

[41] Nagios, "Nagios: The industry standard in it infrastructure monitoring," https://www.nagios.com/.

[42] G. C. Platform, "Stackdriver-metrics list," https://cloud.google.com/monitoring/api/metrics.

[43] "Stackdriver trace," https://cloud.google.com/trace/docs/overview.

[44] "Using stackdriver trace with zipkin," https://cloud.google.com/trace/docs/zipkin.

[45] SolarWinds, "Solarwinds: The hybrid it monitoring," http://www.solarwinds.com/.

[46] DataDog, "Datadog: Cloud-scale monitoring," https://www.datadoghq.com/.

[47] Sysdig, "Sysdig monitor," https://sysdig.com/.

[48] J. S. Sanz, "Tracking down application bottlenecks with tracers," https://sysdig.com/blog/tracking-down-application-bottlenecks-with-tracers/.

[49] Dynatrace, "Digital performance management for business, operations and development," https://www.dynatrace.com/.

[50] "Collecting and analyzing execution-time data," https://www.dynatrace.com/resources/ebooks/javabook/collecting-execution-time-data/.

[51] N. Relic, "New relic, performance monitoring for today's digital business," https://newrelic.com/.

[52] Zenoss, "zenoss: The hybrid it monitoring platform," https://www.zenoss.com.

[53] L. V. API, "Libvirt-snmp," http://wiki.libvirt.org/page/Libvirt-snmp.

[54] vmware, "Realize hyperic monitors software across all environments," http://www.vmware.com/products/vrealize-hyperic.html.

[55] S. A. D. Chaves, R. B. Uriarte, and C. B. Westphall, "Toward an architecture for monitoring private clouds," *IEEE Communications Magazine*, vol. 49, no. 12, pp. 130–137, December 2011.

[56] O. Platform, "Opennebula," https://opennebula.org/.

[57] "Opennebula-monitoring," https://docs.opennebula.org/5.2/deployment/open_cloud_host_setup/monitoring.html.

[58] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[59] R. A. Vyas, H. B. Prajapati, and V. K. Dabhi, "Embedding custom metric in ganglia monitoring system," in *Advance Computing Conference (IACC), 2014 IEEE International*. IEEE, 2014, pp. 793–797.

[60] N. Z. An, "Ganglia documents," https://github.com/ganglia/monitor-core/wiki/Ganglia-Documents.

[61] M. Massie, B. Li, B. Nicholes, V. Vuksan, R. Alexander, J. Buchbinder, F. Costa, A. Dean, D. Josephsen, P. Phaal *et al.*, *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale*. " O'Reilly Media, Inc.", 2012.

[62] K. N. Masao Yamamoto, Miyuki Ono, "Unified performance profiling of an entire virtualized environment," in *International Journal of Networking and Computing*, January 2016, pp. 124–147.

[63] J. Du, N. Sehrawat, and W. Zwaenepoel, "Performance profiling of virtual machines," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 3–14.

[64] A. Khandual, "Performance monitoring in linux kvm cloud environment," in *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*, Oct 2012, pp. 1–6.

[65] A. Anand, M. Dhingra, J. Lakshmi, and S. K. Nandy, "Resource usage monitoring for kvm based virtual machines," in *Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on*, Dec 2012, pp. 66–70.

[66] S. Wang, W. Zhang, T. Wang, C. Ye, and T. Huang, "Vmon: Monitoring and quantifying virtual machine interference via hardware performance counter," in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, vol. 2. IEEE, 2015, pp. 399–408.

[67] R. Matloobi and A. Y. Zomaya, "Managing performance degradation of collocated virtual machines in private cloud," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec 2016, pp. 128–135.

[68] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut, "Perfcompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1742–1755, June 2016.

[69] C. H. Kim, J. Rhee, K. H. Lee, X. Zhang, and D. Xu, "Perfguard: binary-centric application performance monitoring in production environments," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 595–606.

[70] A. Bala and I. Chana, "Prediction-based proactive load balancing approach through vm migration," *Engineering with Computers*, pp. 1–12, 2016.

[71] X. Zhao, J. Yin, Z. Chen, and S. He, "Workload classification model for specializing virtual machine operating system," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 343–350.

[72] X. Zhao and J. Yin, "vspec: workload-adaptive operating system specialization for virtual machines in cloud computing," *Science China Information Sciences*, vol. 59, no. 9, p. 92105, Aug 2016. [Online]. Available: https://doi.org/10.1007/s11432-015-5387-6

[73] M. Gebai, F. Giraldeau, and M. R. Dagenais, "Fine-grained preemption analysis for latency investigation across virtual machines," *Journal of Cloud Computing*, vol. 3, no. 1, p. 23, Dec 2014. [Online]. Available: https://doi.org/10.1186/s13677-014-0023-3

[74] C. Biancheri and M. R. Dagenais, "Fine-grained multilayer virtualized systems analysis," *Journal of Cloud Computing*, vol. 5, no. 1, p. 19, 2016.

[75] C. Biancheri, N. E. Jivan, and M. R. Dagenais, "Multilayer virtualized systems analysis with kernel tracing," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, Aug 2016, pp. 1–6.

[76] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software techniques for avoiding hardware virtualization exits." in *USENIX Annual Technical Conference*, 2012, pp. 373–385.

[77] F. Messina, G. Pappalardo, D. Rosaci, and G. M. L. Sarné, "An agent based architecture for vm software tracking in cloud federations," in *2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*, July 2014, pp. 463–468.

[78] D. Trihinas, G. Pallis, and M. Dikaiakos, "Monitoring elastically adaptive multi-cloud services," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[79] "iostat," http://linux.die.net/man/1/iostat, accessed: 2018-02-12.

[80] "vmstat," http://linuxcommand.org/man_pages/vmstat8.html, 2005, accessed: 2018-12-12.

[81] J. M. A. Calero and J. G. Aguado, "Monpaas: An adaptive monitoring platformas a service for cloud computing infrastructures and services," *IEEE Transactions on Services Computing*, vol. 8, no. 1, pp. 65–78, Jan 2015.

[82] H. Nemati, A. Singhvi, N. Kara, and M. E. Barachi, "Adaptive sla-based elasticity management algorithms for a virtualized ip multimedia subsystem," in *2014 IEEE Globecom Workshops (GC Wkshps)*, Dec 2014, pp. 7–11.

[83] "Libvmi, virtual machine introspection library," http://libvmi.com/, accessed: 2018-12-25.

[84] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Vmm-based hidden process detection and identification using lycosid," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346269

[85] L. Litty and D. Lie, "Manitou: A layer-below approach to fighting malware," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving*

Software Dependability, ser. ASID '06. New York, NY, USA: ACM, 2006, pp. 6–11. [Online]. Available: http://doi.acm.org/10.1145/1181309.1181311

[86] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vpath: Precise discovery of request processing paths from black-box observations of thread and network activities," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 19–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855807.1855826

[87] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, Aug 2016.

[88] C. Canali and R. Lancellotti, "Exploiting classes of virtual machines for scalable iaas cloud management," in *Journal of Communications Software and Systems*, vol. 8, no. 4, 2012, pp. 102–109.

[89] ——, "Detecting similarities in virtual machine behavior for cloud monitoring using smoothed histograms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 8, pp. 2757 – 2769, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514000343

[90] ——, "Exploiting ensemble techniques for automatic virtual machine clustering in cloud systems," *Automated Software Engineering*, vol. 21, no. 3, pp. 319–344, Sep 2014. [Online]. Available: https://doi.org/10.1007/s10515-013-0134-y

[91] ——, "Exploiting classes of virtual machines for scalable iaas cloud management," in *2015 IEEE Fourth Symposium on Network Cloud Computing and Applications (NCCA)*, June 2015, pp. 15–22.

[92] M. Abdelsalam, R. Krishnan, and R. Sandhu, "Clustering-based iaas cloud monitoring," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, June 2017, pp. 672–679.

[93] M. Abdelsalam, R. Krishnan, Y. Huang, and R. Sandhu, "Malware detection in cloud infrastructures using convolutional neural networks," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 162–169. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00028

[94] X. Zhang, F. Meng, and J. Xu, "Perfinsight: A robust clustering-based abnormal behavior detection system for large-scale cloud," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018, pp. 896–899.

[95] H. Zhang, J. Rhee, N. Arora, S. Gamage, G. Jiang, K. Yoshihira, and D. Xu, "Clue: System trace analytics for cloud service performance diagnosis," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, May 2014, pp. 1–9.

[96] A. Abusitta, M. Bellaiche, and M. Dagenais, "An svm-based framework for detecting dos attacks in virtualized clouds under changing environment," *Journal of Cloud Computing*, vol. 7, no. 1, p. 9, Apr 2018. [Online]. Available: https://doi.org/10.1186/s13677-018-0109-4

[97] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093 – 2115, 2013.

[98] "Linux kenrel github," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/arch/x86/kvm/vmx.c, 2018, accessed: 2018-12-12.

[99] "Support for running esxi/esx as a nested virtualization solution," https://kb.vmware.com/s/article/2009916, 2017, accessed: 2018-12-12.

[100] "Nested virtualization in xen," https://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen, 2017, accessed: 2018-12-12.

[101] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 203–216. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043576

[102] "Amazon cloudwatch," https://aws.amazon.com/cloudwatch/, 2018, accessed: 2018-12-12.

[103] "Cisco cloud consumption service," http://www.cisco.com/c/en/us/solutions/cloud/cloud-consumption-service.html, 2016, accessed: 2018-12-12.

[104] "OpenStack telemetry measurements metrics definition," http://docs.openstack.org/admin-guide-cloud/telemetry-measurements.html, 2017, accessed: 2018-12-12.

[105] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *USENIX Conference on Annual Technical Conference*. USENIX Association, 2013, pp. 219–230.

[106] S. Wang, W. Zhang, T. Wang, C. Ye, and T. Huang, "Vmon: Monitoring and quantifying virtual machine interference via hardware performance counter," in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, vol. 2, July 2015, pp. 399–408.

[107] M. Graziano, A. Lanzi, and D. Balzarotti, *Hypervisor Memory Forensics*, 2013, pp. 21–40.

[108] C. Biancheri, N. Eezzati-Jivan, and M. R. Dagenais, "Multilayer virtualized systems analysis with kernel tracing," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, Aug 2016, pp. 1–6.

[109] H. Nemati and M. R. Dagenais, "Virtual cpu state detection and execution flow analysis by host tracing," in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, Oct 2016, pp. 7–14.

[110] S. D. Sharma, H. Nemati, G. Bastien, and M. Dagenais, "Low overhead hardware-assisted virtual machine analysis and profiling," in *2016 IEEE Globecom Workshops (GC Wkshps)*, Dec 2016, pp. 1–6.

[111] H. Nemati, S. D. Sharma, and M. R. Dagenais, "Fine-grained nested virtual machine performance analysis through first level hypervisor tracing," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 84–89. [Online]. Available: https://doi.org/10.1109/CCGRID.2017.20

[112] "Open source software for creating clouds ," https://www.openstack.org/, 2018, accessed: 2018-12-12.

[113] "Who's using redis," http://redis.io/topics/whos-using-redis, 2018, accessed: 2018-02-12.

[114] H. Nemati and M. R. Dagenais, "virtflow: Guest independent execution flow analysis across virtualized environments," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2018.

[115] R. Willis, "Critical path analysis and resource constrained project scheduling — theory and practice," *European Journal of Operational Research*, vol. 21, no. 2, pp. 149 – 155, 1985.

[116] H. Nemati and M. R. Dagenais, "Vm processes state detection by hypervisor tracing," in *2018 Annual IEEE International Systems Conference (SysCon)*, April 2018.

[117] H. Nemati, G. Bastieny, and M. R. Dagenaisz, "Wait analysis of virtual machines using host kernel tracing," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2018, pp. 1–6.

[118] "Amazon i/o characteristics and monitoring," http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-io-characteristics.html, accessed: 2018-12-03.

[119] "Azure resource health overview," https://docs.microsoft.com/en-us/azure/service-health/resource-health-overview, 2018, accessed: 2018-12-12.

[120] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 18:1–18:20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1656980.1657005

[121] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in mirage, an integrated software upgrade testing and distribution system," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 221–236. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294283

[122] "Voip qos requirements," https://www.voip-info.org, 2018, accessed: 2018-12-12.

[123] "The third generation partnership project." http://www.3gpp.org/, accessed: 2018-12-05.

[124] Spirent, in *IMS Architecture: The LTE User Equipment Perspective*, January 2018.

[125] M. Homayouni, H. Nemati, V. Azhari, and A. Akbari, "Controlling overload in sip proxies: An adaptive window based approach using no explicit feedback," in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, Dec 2010, pp. 1–5.

[126] "Vm workload database," https://github.com/Nemati/VMClassificationDataBase, accessed: 2018-11-03.

[127] S. V. A. Hani Nemati and M. R. Dagenais, "Host hypervisor trace mining for virtual machine workload characterization," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, June 2019.

[128] "(kvm) kernel virtual machine," http://www.linux-kvm.org/page/Main_Page, accessed: 2019-01-30.

[129] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, and M. R. Dagenais, "State history tree: An incremental disk-based data structure for very large interval data," in *2013 International Conference on Social Computing*, Sep. 2013, pp. 716–724.

[130] J. Macqueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Mathematical Statist. Probability*, 1967, pp. 281–297.

[131] I. S. Dhillon, Y. Guan, and B. Kulis, "Kernel k-means: Spectral clustering and normalized cuts," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '04. New York, NY, USA: ACM, 2004, pp. 551–556. [Online]. Available: http://doi.acm.org/10.1145/1014052.1014118